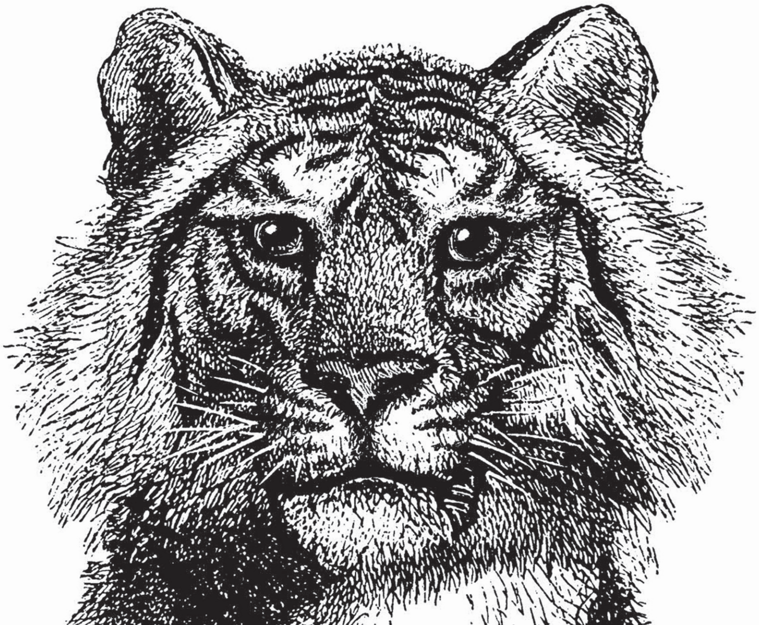
O'REILLY

TURING 图灵程序设计丛书

第6版
涵盖Java 8



Java
技术手册

Java in a Nutshell

[英] Benjamin J. Evans [美] David Flanagan 著
安道 译

中国工信出版集团 人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

译者介绍



安道

专注于现代化计算机技术的自由译者，译有《Flask Web开发》《Python网络编程攻略》《JavaScript Web应用开发》《Ruby on Rails教程》《Python Web开发》等书。个人网站：http://about.ac。

TURING 图灵程序设计丛书

Java技术手册（第6版）

Java in a Nutshell

Sixth Edition

［英］Benjamin J. Evans［美］David Flanagan 著
安道 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北京

图书在版编目（C I P）数据

Java技术手册 ：第6版 / （英）埃文斯
（Evans,B.J.），（美）弗拉纳根（Flanagan,D.）著 ；安
道译. -- 北京 ：人民邮电出版社，2015.11
（图灵程序设计丛书）
ISBN 978-7-115-40609-5

Ⅰ. ①J… Ⅱ. ①埃… ②弗… ③安… Ⅲ. ①JAVA语
言－程序设计－技术手册 Ⅳ. ①TP312-62

中国版本图书馆CIP数据核字(2015)第245078号

内 容 提 要

本书为《Java 技术手册》的升级版，涵盖最新的 Java 7 和 Java 8。第一部分介绍 Java 编程语言和 Java 平台，主要内容有 Java 环境、Java 基本句法、Java 面向对象编程、Java 类型系统、Java 的面向对象设计、Java 实现内存管理和并发编程的方式。第二部分通过大量示例来阐述如何在 Java 环境中完成实际的编程任务，主要内容有编程和文档约定，使用 Java 集合和数组，处理常见的数据格式，处理文件和 I/O，类加载、反射和方法句柄，Nashorn，以及平台工具和配置。

本书适用于有经验的 Java 开发人员掌握 Java 7 和 Java 8，也适合新手开发人员学习。

◆ 著　　[英] Benjamin J. Evans
　　　　[美] David Flanagan
译　　安 道
责任编辑　岳新欣
执行编辑　曹静雯　李敏
责任印制　杨林杰

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：800×1000　1/16
印张：22
字数：520千字　　　　2015年 11 月第 1 版
印数：1－4 000册　　　2015年 11 月北京第 1 次印刷

著作权合同登记号　图字：01-2015-5423号

定价：79.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京崇工商广字第 0021 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

献给所有传播和平及抵制暴力的人。

目录

第二部分 使用 Java 平台

序

2013 年至 2014 年的冬天，英国一直饱受着异常猛烈的暴风雪的侵袭。风暴使一些沉船浮出水面，随之而来的还有一些惊人的考古发现。我的家乡康沃尔郡更是有重大发现，其中最引人注目的是一片石化森林，可以追溯到最近的一次冰川期的末期，如今已被海水和沙滩覆盖。我很幸运，在海水再次将其淹没之前的低潮期，我去看了这片森林，而且还花了几小时勘察。

在残存的树根、树桩以及即将变成泥炭的有机物中，我仍然可以辨认出零星的枝干和树皮。我漫步在潮涨潮落的海滩上，无意中发现了半个坚果。这种坚果树早已不在此纬度地区生长，虽然外面有一层有机物，但坚果的形状依稀可见，而且可以看出它能存活很长一段时间。

在对 David 的这本经典著作进行修订时，我希望能彰显出那棵史前坚果树的精神。如果既能保留前一版的良好结构以及简洁的风格，又能引起新一代开发者的关注，突出重点，我就十分满足了。

Ben Evans
2014 年

前言

这是一本 Java 案头参考书，适合放在键盘旁，编程时随时翻阅。本书第一部分快速准确地介绍 Java 编程语言和 Java 平台的核心运行时概念。第二部分通过重要的核心 API 示例来解释关键概念。本书虽然涵盖 Java 8，但考虑到有些行业还没有开始使用，所以只要有需要，我们就会特别注明 Java 8（部分是 Java 7）引入的功能。本书全面使用 Java 8 句法，以前可能使用匿名嵌套类的地方会换用 lambda 表达式。

第6版的变化

本书第 5 版涵盖 Java 5，而这一版涵盖 Java 8。本书第 5 版大约在十年前出版，在那之后，Java 语言和程序员的工作环境都发生了重大变化。因此，这一版内容变化很大。其中一个重要的变化是，不再像前几版那样对平台的核心 API 进行详尽的介绍。

其中一个原因是，在纸质书中印出数量巨大的核心 API 是不切实际的。而一个更有说服力的原因是，随时可连的快速互联网不断普及，几乎所有 Java 程序员工作时都会连接互联网。详细的 API 文档参考更适合放在网上，而不是印在书中。

相应地，这一版删掉了占据第 5 版三分之二篇幅的 API 参考，剩下的内容才符合“概要型”手册的要求。当代 Java 开发者不仅需要了解句法和 API，当 Java 环境成熟后，并发、面向对象设计、内存管理和 Java 类型系统这些话题都变得重要了，主流开发者都要了解。

我们在这一版中试图反映出 Java 生态系统这些年的变化，因此很大程度上摒弃了前几版的写作方式。具体而言，我们基本不会详细说明某个 Java 特性是在哪个版本中引入的，因为大多数 Java 开发者只关心最新版。

本书内容

本书前 6 章介绍 Java 语言和 Java 平台——这些内容一定要仔细阅读。这本书偏向 Oracle/

OpenJDK（Open Java Development Kit）对 Java 的实现，但又不局限于此，使用其他 Java 环境的开发者仍然能看到很多对其他环境的介绍。第一部分包括如下内容。

- 第 1 章　Java 环境介绍

  这一章概述 Java 语言和 Java 平台，说明 Java 的重要特性和优势，包括 Java 程序的生命周期。最后会介绍 Java 的安全性，并回应一些针对 Java 的批评。

- 第 2 章　Java 基本句法

  这一章详细介绍 Java 编程语言，包括 Java 8 的改动。这一章内容很多，也很详细，不过阅读前不需要读者有大量编程经验。有经验的 Java 程序员可以把这一章当成语言参考。有大量 C 和 C++ 开发经验的程序员阅读这一章之后，也能快速了解 Java 的句法。只有少量编程经验的初学者经过认真阅读，应该也能学会 Java 编程，不过最好再结合其他资料一起学习，例如 Bert Bates 和 Kathy Sierra 合著的 *Head First Java*（O'Reilly 出版，http://shop.oreilly.com/product/9780596009205.do）。

- 第 3 章　Java 面向对象编程

  这一章介绍如何利用第 2 章介绍的 Java 基本句法，使用类和对象编写简单的面向对象程序。这一章章不要求读者有面向对象编程经验。新手程序员可以将其当成教程，有经验的 Java 程序员则可以当作参考。

- 第 4 章　Java 类型系统

  这一章以前面对 Java 面向对象编程的说明为基础，介绍 Java 类型系统的其他方面，例如泛型、枚举类型和注解。全面地了解类型系统之后，我们就可以讨论 Java 8 最大的变化了——lambda 表达式。

- 第 5 章　Java 的面向对象设计

  这一章概述设计可靠的面向对象程序所需的一些基本技术，还会简单介绍一些设计模式及其在软件工程中的用处。

- 第 6 章　Java 实现内存管理和并发编程的方式

  这一章讨论 Java 虚拟机代替程序员管理内存的方式，以及内存、可见性与 Java 并发编程和线程之间错综复杂的关系。

前 6 章教你如何使用 Java 语言，也介绍了 Java 平台最重要的概念。本书第二部分则告诉你如何在 Java 环境中完成实际的编程任务。这部分包含大量示例，以攻略方式撰写。第二部分包括如下内容。

- 第 7 章　编程和文档约定

  这一章介绍 Java 编程中重要且运用广泛的重要约定，还会介绍如何使用特定格式的文档注释来让 Java 代码进行自我文档化。

- 第 8 章　使用 Java 集合

  这一章介绍 Java 的标准集合库，包含几乎对每个 Java 程序都很重要的数据结构，例如 `List`、`Map` 和 `Set`。此外，还会详细介绍新引入的 `Stream` 抽象，以及 lambda 表达式和集合之间的关系。

- 第 9 章　处理常见的数据格式

  这一章说明如何有效使用 Java 处理常见的数据类型，例如文本、数字和时间相关的信息（日期和时间）。

- 第 10 章　处理文件和 I/O

  这一章涵盖几种不同的文件处理方式，包括 Java 旧版中的经典方式和现代化的异步方式，最后还会简单介绍如何使用 Java 平台的核心 API 处理网络。

- 第 11 章　类加载、反射和方法句柄

  这一章介绍 Java 隐含的元编程功能。首先介绍 Java 类型元信息的概念；然后介绍类加载，以及 Java 的安全模型和动态类型加载之间的关系；最后，介绍几个类加载程序和相对较新的方法句柄功能。

- 第 12 章　Nashorn

  这一章介绍 Nashorn，这是一个运行在 Java 虚拟机中的 JavaScript 实现。Nashorn 在 Java 8 中引入，它在其他 JavaScript 实现之外又提供了一个选择。这一章末尾会介绍 Avatar.js，这是和 Node 兼容的服务器端技术。

- 第 13 章　平台工具和配置

  甲骨文提供的 JDK（和 OpenJDK）包含很多有用的 Java 开发工具，其中最重要的是 Java 解释器和编译器。这一章会介绍这些工具。这一章后半部分介绍紧凑配置——这是 Java 8 的新功能，用于精简 Java 运行时环境（Java Runtime Environment，JRE），能显著减少占用空间。

相关书籍

O’Reilly 出版了一个系列的 Java 编程书籍，其中有几本与本书配套，如下所示。

- Pat Niemeyer 和 Daniel Leuck 合著的 *Learning Java*（http://shop.oreilly.com/product/0636920023463.do）

  这是一本全面的 Java 教程，包含 XML 和客户端 Java 编程。

- Richard Warburton 编写的《Java 8 函数式编程》（http://shop.oreilly.com/product/0636920030 713.do）

这本书详细介绍了 Java 8 引入的 lambda 表达式，而且介绍了使用 Java 早期版本的程序员可能不熟悉的函数式编程。

- Bert Bates 和 Kathy Sierra 合著的 *Head First Java*（http://shop.oreilly.com/product/9780596009205.do）

  这本书使用独特的方式介绍 Java。习惯形象化思维的开发者往往觉得这本书是对传统 Java 书籍的补充。

O'Reilly 出版的所有 Java 书籍可以在 http://java.oreilly.com/ 找到。

在线示例

本书中的示例可在本书主页下载，地址是 http://www.oreilly.com/catalog/javanut6。访问这个地址还能看到重要的说明或勘误。

排版约定

本书使用了下述排版约定。

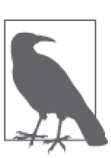
- 楷体
  表示新术语。

- 等宽字体（`Constant width`）
  表示 Java 代码，也表示编程时输入的字面量，例如关键字、数据类型、常量、方法名、变量、类名和接口名。

- 斜体等宽字体（***`Constant Width Italic`***）
  表示函数的参数名称，一般还表示占位符，表明要换成程序中真正使用的值。有时还用来指代概念区域或代码行，例如 ***`statement`***。



这个图标表示提示或建议。



这个图标表示一般性说明。

这个图标表示警告或提醒。

发表评论

请把评论、勘误和建议通过电子邮件地址 javanut6@gmail.com 直接发给本书作者。

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

http://shop.oreilly.com/product/0636920030775.do。

对于本书的评论和技术性问题，请发送电子邮件到：

bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

Safari® Books Online

Safari Books Online（http://www.safaribooksonline.com）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。技术专家、软件开发人员、Web 设计师、商务人士和创意专家等，在开展调研、解决问题、学

习和认证培训时，都将 Safari Books Online 视作获取资料的首选渠道。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

致谢

Meghan Blanchette 是第 6 版的编辑，她十分注重细节、性格开朗、专业扎实，在撰写本书过程中的每个重要时刻都为我们提供了额外的动力。

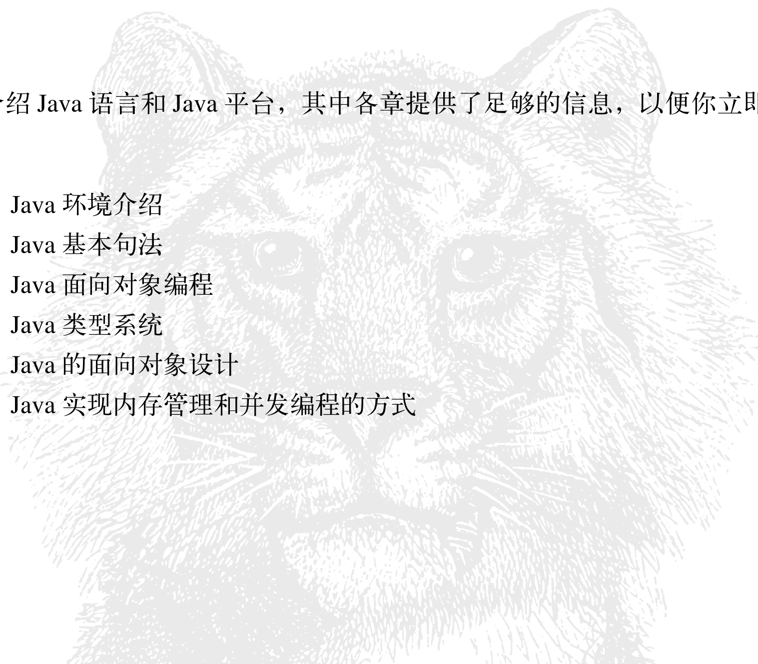
特别感谢 Jim Gough、Richard Warburton、John Oliver、Trisha Gee 和 Stephen Colebourne。

Martijn Verburg 始终是我们的好朋友和商业伙伴，我们经常征求他的意见，他为我们提供了很多有用的建议。

Ben 要特别感谢每一个给他反馈和帮助他提升写作能力的人。特别要指出的是 Caroline Kvitka、Victor Grazi、Tori Weildt 和 Simon Ritter，感谢他们提出的有用建议。如果他没有采纳他们的出色建议，当然应该责怪他自己。

第一部分

Java介绍



第一部分介绍 Java 语言和 Java 平台，其中各章提供了足够的信息，以便你立即开始使用 Java。

第 1 章

Java环境介绍

欢迎学习 Java 8。也许应该说欢迎你回来。你可能是从其他语言转到这个生态系统的，也可能这是你学习的第一门编程语言。不管你是如何到达这里的，都要欢迎你。很高兴你选择了 Java。

Java 是一个强大且通用的编程环境，是世界上使用范围最广的编程语言之一，在商务和企业计算领域取得了极大的成功。

本章介绍 Java 语言（供程序员编写应用）、Java 虚拟机（用来运行应用）和 Java 生态系统（为开发团队提供很多有价值的编程环境）。

我们会先简要介绍 Java 语言和虚拟机的历史，然后说明 Java 程序的生命周期，最后厘清 Java 和其他环境之间一些常见的疑问。

本章最后会介绍 Java 的安全性，还会讨论一些安全编程相关的话题。

1.1　Java语言、JVM和生态系统

Java 编程环境出现于 20 世纪 90 年代末，由 Java 语言和运行时组成。运行时也叫 Java 虚拟机（Java Virtual Machine，JVM）。

Java 刚出现时，这种分离方式很新奇，但最近软件开发的趋势表明，这已经变成了通用做法。值得一提的是微软的 .NET 环境，它比 Java 晚几年出现，但沿用了非常类似的平台架构方式。

微软的 .NET 平台和 Java 相比有个重要的区别，人们都觉得 Java 是相对开放的生态系统，有多个开发方。在 Java 的演进过程中，这些开发方在合作的同时也有竞争，不断推进 Java 的技术发展。

Java 成功的主要原因之一是，整个生态系统是个标准的环境。这意味着组成 Java 环境的各种技术都有规范。这些标准让开发者和客户相信，自己所用的技术能和其他组件兼容，即便来自不同的技术提供方也不怕。

Java 目前归甲骨文公司所有（甲骨文收购了发明 Java 的太阳计算机系统公司，以下简称 Sun）。红帽、IBM、惠普、SAP、苹果和富士通等公司也大量参与了 Java 标准技术的实现。

Java 也有开源版本，叫 OpenJDK，由多家公司合作开发。

其实，Java 由多个不同但相互联系的环境和规范组成，包括 Java 移动版（Java ME）、Java 标准版（Java SE）和 Java 企业版（Java EE）。本书只涵盖 Java SE 第 8 版。

后面会详细说明标准，现在先介绍 Java 语言和 JVM。这是两个不同但互有关联的概念。

1.1.1 Java语言是什么

Java 程序的源码使用 Java 语言编写。Java 是人类可读的编程语言，基于类，而且面向对象，比较易读易写（偶尔有点啰嗦）。

Java 有意识地降低了教、学成本，参考了 C++ 等语言的行业经验，尽量删除了复杂的功能，但保留了“前辈”编程语言的精粹。

总的来说，Java 的目的是为企业开发商业应用提供坚实稳定的基础。

作为一门编程语言，Java 的设计相对保守，而且改动频率低。这么做是有意保护企业对 Java 技术的投入。

Java 语言自 1996 年发布之后，一直在不断地修订（但没有完全重写）。也就是说，一开始为 Java 选择的设计方式，即 20 世纪 90 年代末采用的那些权宜之计，现在仍旧影响着这门语言。详情参见第 2 章和第 3 章。

Java 8 的变动幅度很大，是近十年来罕见的（有些人觉得是 Java 出现以来最大的变动）。lambda 表达式的引入和核心中集合 API 的大幅度改写等，将彻底改变大多数 Java 开发者编写代码的方式。

Java 语言受 Java 语言规范（Java Language Specification，JLS）的约束，这个规范限定了某项功能必须采用某种方式实现。

1.1.2 JVM是什么

JVM 是一个程序，提供了运行 Java 程序所需的运行时环境。如果某个硬件和操作系统平台没有相应的 JVM，就不能运行 Java 程序。

幸好，JVM 被移植到了大多数设备中，机顶盒、蓝光播放器、大型机或许都有适用的 JVM。

Java 程序一般都在命令行中启动，例如：

```
java <arguments> <program name>
```

这个命令会在操作系统的一个进程中启动 JVM，提供 Java 运行时环境，然后在刚启动的（空）虚拟机中运行指定的程序。

有一点很重要，你要知道：提供给 JVM 运行的程序不是 Java 语言源码，源码必须转换（或编译）成一种称为 Java 字节码的格式。提供给 JVM 的 Java 字节码必须是类文件格式，其扩展名为 .class。

JVM 是字节码格式程序的解释器，一次只执行字节码中的一个指令。而且，你还要知道，JVM 和用户提供的程序都能派生额外的线程，所以用户提供的程序中可能同时运行着多个不同的函数。

JVM 的设计方式建立在几个早期编程环境的多年发展经验之上，尤其是 C 和 C++，因此有多个目的，这些目的都是为了减轻程序员的负担。

- 包含一个容器，让应用代码在其中运行。
- 较之 C/C++，提供了一个安全的执行环境。
- 代开发者管理内存。
- 提供一个跨平台的执行环境。

介绍 JVM 时往往都会提到这些目的。

前面介绍 JVM 和字节码解释器时已经提到了第一个目的，即 JVM 是应用代码的容器。

第 6 章介绍 Java 环境如何管理内存时会讨论第二个和第三个目的。

第四个目的有时也说成“一次编写，到处运行”，意思是 Java 类文件可从一个运行平台迁移到另一个平台，只要有可用的 JVM，就能正常运行。

也就是说，Java 程序可以在运行着 OS X 的苹果 Mac 电脑中开发（并转换成类文件），然后把类文件移到 Linux 或微软 Windows（或其他平台）中，无需任何改动，Java 程序依然能运行。

> Java 环境被移植到了众多平台中，除了 Linux、Mac 和 Windows 等主流平台外，还支持很多其他平台。本书使用“大多数实现”来概括大多数开发者能接触到的平台。Mac、Windows、Linux、Solaris、BSD Unix 和 AIX 等被视为“主流平台”，都算在“大多数实现”的范围之内。

除了上述四个主要目的之外，JVM 还有一个设计方面的考量很少被提及和讨论，即 JVM 使用运行时信息进行自我管理。

20 世纪 70 年代和 80 年代对软件的研究表明，程序运行时的行为有很多有趣且有用的模式无法在编译时推论得出。JVM 是真正意义上第一个利用这项研究结果的主流平台。

JVM 会收集运行时信息，从而对如何执行代码做出更好的决定。也就是说，JVM 能监控并优化运行在其中的程序，而没有这种能力的平台则做不到这一点。

一个典型的例子是，在运行 Java 程序的生命周期中，各组成部分被调用的次数并不都是相同的，有些部分调用的次数远比其他部分多得多。Java 平台使用一种名为 JIT 编译（just-in-time compilation）的技术解决这个问题。

在 HotSpot JVM（Sun 为 Java 1.3 开发的 JVM，现在仍在使用）中，JVM 首先识别程序的哪一部分调用最频繁（这一部分叫“热点方法”），然后跳过 JVM 解释器，直接把这一部分编译成机器码。

JVM 利用可用的运行时信息，让程序的性能比纯粹经解释器执行更高。事实上，很多情况下，JVM 使用的优化措施得到的性能提升，已经超过了编译后的 C 和 C++ 代码。

描述 JVM 必须怎样运行的标准叫 JVM 规范。

1.1.3 Java生态系统是什么

Java 语言易于学习，而且和其他编程语言相比，拥有的抽象更少。JVM 为 Java 语言（或其他语言）的运行提供了坚实的基础，并且它写出的程序性能高且是可移植的。这两种相互联系的技术放在一起，可以让企业放心选择在何处下力发展。

然而，Java 的优势不止于此。自 Java 初期开始，就形成了范围极广的生态系统，里面有大量的第三方库和组件。也就是说，开发团队能从现有的连接器和驱动器中获益良多，从中他们能获得几乎任何能想到的技术，有些收费，有些则开源。

在当下的技术生态系统中，很少出现某个技术组件不提供 Java 连接器的情况。不管是传统的关系数据库，还是 NoSQL，或者各种企业级监控系统和消息系统，都能集成到 Java 中。

这些正是企业和大型公司采用 Java 技术的主要驱动力。使用现有的库和组件能释放开发团队的潜能，让开发者作出更好的选择，利用 Java 核心技术实现最佳的开放式架构。

1.2　Java和JVM简史

- Java 1.0（1996年）

  这是 Java 的第一个公开发行版，只包含 212 个类，分别放在八个包中。Java 平台始终关注向后兼容性，所以使用 Java 1.0 编写的代码，不用修改或者重新编译，依旧能在最新的 Java 8 中运行。

- Java 1.1（1997年）

  这一版 Java 平台是原来的两倍多，并且引入了“内部类”和第一版反射 API。

- Java 1.2（1998年）

  这是 Java 一个非常重要的版本。这一版 Java 平台是原来的三倍，而且首次出现了集合 API（包括 `Set`、`Map` 和 `List`）。1.2 版增加的新功能过多，Sun 不得不把平台重新命名为“Java 2 Platform”。这里的“Java 2”是商标，而不是真实的版本号。

- Java 1.3（2000年）

  这其实是个维护版本，主要用于修正缺陷，解决稳定性，并提升性能。这一版还引入了 HotSpot Java 虚拟机，这个虚拟机现在还在使用（不过有大量的修改和改进）。

- Java 1.4（2002年）

  这也是一个重要的版本，增加了一些重要的功能，例如高性能低层 I/O API、处理文本的正则表达式、XML 和 XSLT 库、SSL 支持、日志 API 和加密支持。

- Java 5（2004年）

  这一版 Java 更新幅度很大，对核心语言做了很多改动，引入了泛型、枚举类型（enum）、注解、变长参数方法、自动装包和新版 `for` 循环。改动的量非常大，所以不得不修改主版本号，以新的主版本号发布。这一版包含 3562 个类和接口，分别放在 166 个包中。在增加的内容中，值得一提的有并发编程的实用工具、远程管理框架和类，以及 Java 虚拟机本身的监测程序。

- Java 6（2006年）

  这一版也主要是维护和提升性能，引入了编译器 API，扩展了注解的用法和适用范围，还提供了绑定，允许脚本语言和 Java 交互。这一版还对 JVM 和 Swing GUI 技术进行了缺陷修正和改进。

- Java 7（2011年）

  这是甲骨文公司接管 Java 后发布的第一个版本，包含语言和平台的多项重要升级。这一版引入了处理资源的 `try` 语句和 NIO.2 API，让开发者编写的资源和 I/O 处理代码更安全且不易出错。方法句柄 API 是反射 API 的替代品，更简单也更安全，而且打开了动态调用（`invokedynamic`）的大门（Java 1.0 之后第一种新字节码）。

- Java 8（2014年）

  这是最新版 Java，变动的幅度是自 Java 5（甚至可能是自 Java 出现）以来最大的一次。这一版引入的 lambda 表达式有望显著提升开发者的效率；集合 API 也升级了，改用 lambda 实现，为此，Java 的面向对象实现方式也发生了根本性变化。其他重要更新包括：实现运行在 JVM 中的 JavaScript（Nashorn），新的日期和时间支持，以及 Java 配置（用于生成不同版本的 Java，尤其适合部署无界面或服务器应用）。

1.3　Java程序的生命周期

为了更好地理解 Java 代码是怎么编译和执行的，以及 Java 和其他编程环境的区别，请看图 1-1 中的流程图。

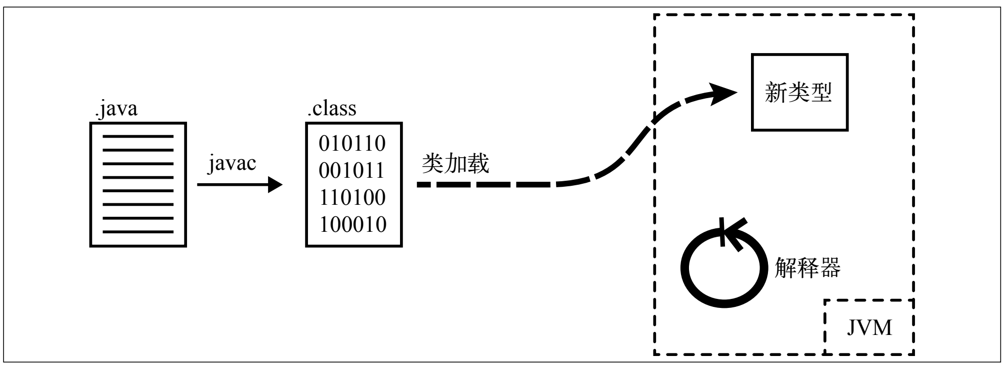


图 1-1：Java 代码是怎么编译和加载的

整个流程从 Java 源码开始，经过 `javac` 程序处理后得到类文件，这个文件中保存的是编译源码后得到的 Java 字节码。类文件是 Java 平台能处理的最小功能单位，也是把新代码传给运行中程序的唯一方式。

新的类文件通过类加载机制载入虚拟机（有关类加载机制更详细的说明参见第 10 章），从而把新类型提供给解释器执行。

常见问题解答

本节解答一些关于 Java 和在 Java 环境中编写的程序的生命周期的最常见问题。

1. 字节码是什么

开发者首次接触 JVM 时，可能认为它是“电脑中的电脑”，然后顺其自然把字节码理解为“内部电脑中 CPU 执行的机器码”或“虚拟处理器执行的机器码”。

其实，字节码和运行于硬件处理器中的机器码不太一样。计算机科学家视字节码为一种

“中间表现形式”，处在源码和机器码之间。

字节码的目的是，提供一种能让 JVM 解释器高效执行的格式。

2. javac是编译器吗

编译器一般生成机器码，而 javac 生成的是和机器码不太一样的字节码。不过，类文件有点像对象文件（例如 Windows 中的 .dll 文件，或 Unix 中的 .so 文件），人类肯定读不懂。

在计算机科学理论的术语中，javac 非常像编译器的“前半部分”，它生成的中间表现形式可以进一步处理，生成机器码。

不过，因为类文件的生成是构建过程中单独的一步，类似于 C/C++ 中的编译，所以很多开发者都把运行 javac 的操作称为编译。在本书里，我们使用术语“源码编译器”或“javac 编译器”表示生成类文件的 javac。

我们把“编译”看作一个单独的术语，表示 JIT 编译，因为只有 JIT 编译才会生成机器码。

3. 为什么叫“字节码”

指令码（操作码）只占一个字节（有些操作还可以有参数，即跟随其后的字节流），所以只有 256 个可用的指令。实际上，有些指令用不到，大概只会使用 200 个，而且其中还有一些是最新版 javac 不支持的。

4. 字节码是优化过的吗

Java 平台的早期阶段，javac 会对生成的字节码进行大量优化。后来表明这么做是错的。JIT 编译出现后，重要的方法会被编译成运行速度很快的机器码。之所以要减轻 JIT 编译器的负担，是因为 JIT 编译获得的效果，比字节码优化多很多，而且字节码还要经过解释器处理。

5. 字节码真的与设备无关吗？那字节顺序呢

不管在哪种设备中生成，字节码的格式都是一样的，其中也包括设备使用的字节顺序。如果你想知道，我告诉你，字节码始终使用大字节序（big-endian）。

6. Java是解释性语言吗

JVM 基本上算是解释器（通过 JIT 编译大幅提升性能）。可是，大多数解释性语言（例如 PHP、Perl、Ruby 和 Python）都直接从源码解释程序（一般会从输入的源码文件中构建一个抽象句法树）。而 JVM 解释器需要的是类文件，因此当然需要多一步操作，即使用 javac 编译源码。

7. 其他语言可以在JVM中运行吗

可以。JVM 可以运行任何有效的类文件，因此，Java 之外的语言可以通过两种方式在 JVM 中运行。第一种，提供用于生成类文件的源码编译器（类似于 javac），以类似 Java 代码的方式在 JVM 中运行（Scala 等语言采用的是这种方式）。

Java 之外的语言可以使用 Java 实现解释器和运行时，然后解释该语言使用的源码格式。JRuby 等语言采用的就是这种方式（不过 JRuby 的运行时很复杂，某些情况下能辅助 JIT 编译）。

1.4 Java的安全性

Java 的设计始终考虑安全性，因此和很多其他现有系统和平台相比有很大的优势。Java 的安全架构由安全专家设计，而且这个平台发布之后，很多其他安全专家仍在研究和探讨。专家们一致认为，Java 的安全架构坚固牢靠，在设计层面没有任何安全漏洞（至少还没有发现）。

Java 安全模型的基础是，严格限制字节码能表述的操作，例如，不能直接访问内存，因此避免了困扰 C 和 C++ 等语言的一整类安全问题。而且，只要 JVM 加载了不信任的类，就会执行字节码校验操作，从而避免了大量问题（字节码校验的更多信息，参见第 10 章）。

尽管如此，没有任何系统能保证 100% 的安全性，Java 也不例外。

虽然从理论上讲，设计是牢固的，但安全架构的实现是另外一回事，在某些 Java 实现中，一直都在发现和修补安全缺陷。

不得不说，Java 8 的延期发布，至少部分原因是发现了一些安全问题，必须要投入时间进行修复。

我相信，在实现 Java 虚拟机的过程中始终都会发现（并修正）安全缺陷。

不过，值得注意的是，最近发现的 Java 安全问题大都与桌面技术有密切联系。在日常的服务器端编程方面，Java 仍是当前最安全的通用平台。

1.5 Java和其他语言比较

本节简要列出 Java 平台和其他你可能熟悉的编程环境之间的重要不同点。

1.5.1 Java和C语言比较

- Java 面向对象，C 面向过程。
- Java 通过类文件实现可移植性，C 需要重新编译。
- Java 为运行时提供了全面的监测程序。
- Java 没有指针，也没有指针相等性运算。
- Java 通过垃圾回收提供了自动内存管理功能。
- Java 无法从低层布局内存（没有结构体）。
- Java 没有预处理器。

1.5.2　Java和C++比较

- Java 的对象模型比 C++ 简单。
- Java 默认使用虚分派（virtual dispatch）。
- Java 始终使用值传递（不过 Java 中的值也能作为对象引用）。
- Java 不完全支持多重继承。
- Java 的泛型没 C++ 的模板强大（不过危害性较小）。
- Java 无法重载运算符。

1.5.3　Java和PHP比较

- Java 是静态类型语言，PHP 是动态类型语言。
- Java 有 JIT，PHP 没有（PHP 6 可能会有）。
- Java 是通用语言，PHP 在网站技术之外很难见到。
- Java 支持多线程，PHP 不支持。

1.5.4　Java和JavaScript比较

- Java 是静态类型语言，JavaScript 是动态类型语言。
- Java 使用基于类的对象，JavaScript 使用基于原型的对象。
- Java 提供了良好的对象封装，JavaScript 没有提供。
- Java 有命名空间，JavaScript 没有。
- Java 支持多线程，JavaScript 不支持。

1.6　回应对Java的一些批评

Java 出现在公共视线中已有很长一段时间了，因此，在这些年里受到的批评也相当多。这些批评可以归咎于一些技术缺点，以及第一版过度的市场推广。

不过，有些批评只是技术圈的传言，不是很准确。本节，我们来看一些常见的抱怨，以及它们在最新版 Java 平台中的状况。

1.6.1　过度复杂

人们经常批评 Java 核心语言过度复杂。即便是 `Object o = new Object();` 这样简单的语句，也有重复——赋值符号左右两边都出现了类型 `Object`。批评人士认为这么做完全是多余的，其他语言都不需要重复声明类型，而且很多辅助功能都不用这么做（例如类型推导）。

这样的说法我不认同。从一开始，Java 的设计目标就是易于阅读（读代码的次数比写代码多很多），许多程序员，尤其是新手，都觉得额外的类型信息有助于阅读代码。

Java 广泛用于企业环境，开发团队往往和运维团队不同。这些额外的信息一般会在处理停机，或者需要维护和修订早就投身其他事务的开发者编写的代码时提供重大帮助。

在最近几个 Java 版本中（7 和后面的版本），语言的设计者已经在尝试回应这些观点，他们寻找可以简化句法复杂度的地方，也更充分地利用类型信息。例如：

```
// 文件辅助方法
byte[] contents =
  Files.readAllBytes(Paths.get("/home/ben/myFile.bin"));

// 使用菱形句法表示重复的类型信息
List<String> l = new ArrayList<>();

// lambda表达式,简化了Runnable
ExecutorService threadPool = Executors.newScheduledThreadPool(2);
threadPool.submit(() -> { System.out.println("On Threadpool"); });
```

然而，Java 的总体原则是非常缓慢且谨慎地修改语言，所以这些变化可能无法完全让批评者满意。

1.6.2 变化慢

Java 第一版发布至今已经超过 15 年了，而且在那个时候也没经过完整修订。在这段时间里，很多其他语言（例如微软的 C#）都发布了不向后兼容的版本，而 Java 没这么做，因此受到了部分开发者的批评。

而且，最近几年，Java 语言因为没有及时吸收其他语言中常见的功能而受到严厉批评。

Sun（现在是甲骨文）在语言设计上采取了保守方式，是为了尽量避免把成本和不合理功能的外部效应强加在大量的用户群体身上。很多使用 Java 的公司都为这一技术注入了重资，语言设计者要认真负责，不能影响现有的用户和安装群体。

每一个新语言功能都要审慎考虑，不只是新功能本身，还要考虑它会如何影响语言现有的功能。有时，新功能的影响会超过目及之处，而 Java 的使用范围又如此广泛，因此可能有很多地方会产生意料之外的影响。

功能发布后，如果有问题，几乎无法将其删除。Java 有一些不合理的功能（例如终结机制），在不影响安装群体的情况下，根本无法安全地删除。语言设计者认为，在语言演进的过程中必须极为小心。

话虽如此，但 Java 8 引入的新语言功能向前迈出了一大步，回应了最常见的功能缺失抱怨，应该能为开发者提供他们一直诉求的语言特性。

1.6.3　性能问题

现在仍然有人批评 Java 平台的速度慢，而且所有批评都集中在“平台”上，这或许是最不合理的批评了。

Java 1.3 引入了 HotSpot 虚拟机和 JIT 编译器，而且在随后的 15 年里，一直在革新和改进虚拟机及其性能。现在，Java 平台的速度异常快，经常会在流行的框架性能评测中取胜，甚至打败了编译成本地机器码的 C 和 C++。

针对这方面的批评大都是因为陈旧的记忆，因为以前的某段时间 Java 很慢。Java 使用的大型且不规则延展的架构方式可能也加深了人们对性能低下的印象。

然而，事实上，任何大型架构都需要评测、分析和性能调校，才能得到最好的表现，Java 也不例外。

Java 平台的核心（Java 语言和 JVM）不仅现在是，以后也仍将是开发者可用的速度最快的通用环境。

1.6.4　不安全

2013 年，Java 平台出现了几个安全漏洞，导致 Java 8 的发布日期延后了。其实，在此之前就有人批评 Java 的安全漏洞数量众多。

在这些漏洞中，有很多都涉及 Java 系统的桌面和 GUI 组件，不会影响使用 Java 编写的网站或其他服务器端代码。

所有编程平台都会时不时地出现安全问题，而且很多其他语言的安全漏洞不比 Java 少，只是少有人知罢了。

1.6.5　太注重企业

Java 平台在公司和企业的开发者中使用广泛，因此觉得 Java 太注重企业一点也不奇怪。人们认为 Java 缺少面向社区的语言所具有的自由风格。

其实，Java 一直都是，而且以后仍将是社区和免费或开源软件开发所广泛使用的语言。在 GitHub 和其他项目托管网站中，Java 是最受欢迎的。

而且，使用范围最广的 Java 语言是通过 OpenJDK 实现的。而 OpenJDK 本身就是开源项目，其社区充满活力，一直在不断增长。

第 2 章

Java基本句法

本章简练而全面地介绍 Java 句法，主要针对之前有些编程经验但刚接触这门语言的读者，对完全没有编程经验的新手也有一些帮助。如果已经了解 Java，可以把这一章当成语言参考。为了方便学过其他编程语言的读者，本章还对 Java 与 C 和 C++ 进行了比较。

本章先介绍非常低层的 Java 句法，然后以此为基础，介绍高级结构。本章包含以下内容。

- 编写 Java 程序的字符，以及这些字符的编码。
- 组成 Java 程序的字面量、标识符和其他标记。
- Java 能处理的数据类型。
- 在 Java 中把单独的标记放在一起组成复杂表达式的运算符。
- 语句：把表达式和其他语句放在一起组成 Java 代码逻辑块。
- 方法：一系列 Java 语句，有名字，可被其他 Java 代码调用。
- 类：由一系列方法和字段组合而成。类是 Java 程序的核心元素，也是面向对象编程的基础。第 3 章专门介绍类和对象。
- 包：由一系列相关的类组合而成。
- Java 程序：由一个或多个交互的类组成，这些类可能来自一个或多个包。

大多数编程语言的句法都很复杂，Java 也不例外。一般来说，介绍一门语言的某些元素时，难免会提到一些尚未接触的元素。例如，介绍 Java 支持的运算符和语句时，不可避免地要提到对象；类似地，介绍对象时也不能不提 Java 的运算符和语句。在学习 Java 或任何其他语言的过程中，都要这样交叉学习。

2.1 Java程序概览

在详细介绍 Java 句法之前，我们先花点儿时间概述 Java 程序。Java 程序由一个或多个 Java 源码文件（或叫*编译单元*）组成。本章末尾会介绍 Java 文件的结构，并且会讲解如何编译和运行 Java 程序。每个编译单元都以可选的 `package` 声明开始，后面跟着零个或多个 `import` 声明。这些声明指定一个命名空间，编译单元中定义的名称都在这个命名空间里，而且还指定了编译单元从哪些命名空间中导入名称。2.10 节会介绍 `package` 和 `import` 声明。

在可选的 `package` 和 `import` 声明之后，是零个或多个引用类型定义。第 3 章和第 4 章会介绍各种可用的引用类型，现在你只需要知道，这些往往都是 `class` 或 `interface` 定义。

在引用类型的定义体中有一些成员，例如*字段*、*方法*和*构造方法*。其中，方法是最重要的成员类型。方法是一段由*语句*组成的 Java 代码。

了解这些基本术语之后，下面开始详细介绍 Java 程序的基本句法单元。句法单元经常被称为*词法标记*（lexical token）。

2.2 词法结构

本节说明 Java 程序的词法结构，首先介绍编写 Java 程序的 Unicode 字符集，然后介绍组成 Java 程序的标记，包括注释、标识符、保留字和字面量等。

2.2.1 Unicode字符集

Java 程序使用 Unicode 字符编写。在 Java 程序中，任何地方都能使用 Unicode 字符，包括注释和标识符，例如变量名。7 位 ASCII 字符集只对英语有用，8 位 ISO Latin-1 字符集只对大多数西欧语言有用，而 Unicode 字符集能表示世界上几乎所有常用的书写语言。

> 如果使用不支持 Unicode 的文本编辑器，或者不想强制查看或编辑你代码的程序员使用支持 Unicode 的编辑器，你可以使用特殊的 Unicode 转义序列 \uxxxx，把 Unicode 字符嵌入 Java 程序。Unicode 转义序列由反斜线、小写的字母 u 和四个十六进制字符组成。例如，`\u0020` 是空格，`\u03c0` 是字符 π。

Java 投入了大量时间和工程努力，确保能最好地支持 Unicode。如果业务应用面向全球用户，特别是西方之外的市场，Java 平台是很好的选择。

2.2.2 区分大小写与空白

Java 语言区分大小写，关键字使用小写，而且必须这么用；也就是说，`While` 和 `WHILE` 与

while 关键字不是一回事。类似地，如果在程序中把变量命名为 i，就不能使用 I 引用这个变量。

> 一般来说，通过大小写来区分标识符是非常糟糕的主意。在代码中不要这么做，尤其不要使用和关键字同名但大小写不同的标识符。

Java 会忽略空格、制表符、换行符和其他空白，除非这些符号出现在引号或字符串字面量中。为了易读，程序员一般会使用空白格式化和缩进代码。本书的示例代码会使用一些常用的缩进约定。

2.2.3 注释

注释是使用自然语言编写的文本，供某一程序的人类读者阅读。Java 编译器会忽略注释。Java 支持三种注释。第一种是单行注释，以 // 字符开始，直到行尾结束。例如：

```
int i = 0;  // 初始化循环变量
```

第二种是多行注释，以 /* 字符开始，不管有多少行，直到 */ 字符结束。javac 会忽略 /* 和 */ 之间的所有文本。虽然这种形式一般用于多行注释，但也可以用于单行注释。

这种注释不能嵌套，即 /* */ 中不能再有 /* */。编写多行注释时，程序员经常使用额外的 * 字符，突出注释的内容。下面是个典型的多行注释：

```
/*
 * 首先,连接服务器。
 * 如果尝试连接失败,立即退出。
 */
```

第三种注释是第二种的一个特例。如果注释以 /** 开头，会被当成特殊的文档注释。和普通的多行注释一样，文档注释也以 */ 结尾，而且不能嵌套。如果你编写了一个 Java 类，希望让其他程序员使用，可以直接在源码中嵌入关于这个类和其中每个方法的文档。名为 javadoc 的程序会提取这些文档，经过处理后生成这个类的在线文档。文档注释中可以包含 HTML 标签和 javadoc 能理解的其他句法。例如：

```
/**
 * 把文件上传到Web服务器中。
 *
 * @param file要上传的文件。
 * @return <tt>true</tt>表示上传成功,
 *         <tt>false</tt>表示上传失败。
 * @author David Flanagan
 */
```

第 7 章会详细介绍文档注释的句法，第 13 章会详细介绍 javadoc 程序。

注释可以出现在 Java 程序中的任何标记之间，但不能出现在标记中。注释尤其不能出现在双引号字符串字面量中。字符串字面量中的注释就是这个字符串的一部分。

2.2.4 保留字

以下是 Java 的保留字（它们是 Java 语言句法的一部分，不能用来命名变量和类等）：

```
abstract   const      final       int         public        throw
assert     continue   finally     interface   return        throws
boolean    default    float       long        short         transient
break      do         for         native      static        true
byte       double     goto        new         strictfp      try
case       else       if          null        super         void
catch      enum       implements  package     switch        volatile
char       extends    import      private     synchronized  while
class      false      instanceof  protected   this
```

后文还会见到这些保留字，其中有些是基本类型的名称，有些是 Java 语句的名称，这两种保留字稍后都会进行介绍。还有一些用于定义类和成员，第 3 章会介绍。

注意，虽然 Java 语言不使用 const 和 goto，但它们也是保留字；interface 还有另外一种形式——@interface，用来定义注解类型。有些保留字（尤其是 final 和 default）根据不同的上下文有不同的意义。

2.2.5 标识符

标识符就是 Java 程序中某个部分的名称，例如类、类中的方法和方法中声明的变量。标识符的长度不限，可以包含 Unicode 字符集中的任意字母和数字，但是不能以数字开头。一般来说，标识符不能包含标点符号，不过可以包含 ASCII 字符集中的下划线（_）和美元符号（$），以及 Unicode 字符集中的其他货币符号，例如 £ 和 ¥。

> 货币符号主要用在自动生成的源码中，例如 javac 生成的代码。不在标识符中使用货币符号，可以避免自己的标识符和自动生成的标识符冲突。

按照规定，可以出现在标识符开头和之中的字符由 java.lang.Character 类中的 isJavaIdentifierStart() 和 isJavaIdentifierPart() 方法定义。

以下是合法标识符示例：

```
i    x1    theCurrentTime    the_current_time 獺
```

特别注意，其中有个 UTF-8 标识符——獭。这是一个汉字，英文是“otter”，完全是个合法的 Java 标识符。在主要是由西方人编写的程序中不常见到使用非 ASCII 字符的标识符，但偶尔也有。

2.2.6 字面量

字面量是直接出现在 Java 源码中的值，包括整数、浮点数、单引号中的单个字符、双引号中的字符串，以及保留字 `true`、`false` 和 `null`。例如，以下都是字面量：

```
1    1.0    '1'    "one"    true    false    null
```

2.3 节会详细介绍表示数字、字符和字符串字面量的句法。

2.2.7 标点符号

Java 标记中也有一些是标点符号。Java 语言规范把这些字符分成两类（有点随意）：分隔符和运算符。分隔符有 12 个：

```
(   )   {   }   [    ]

... @ ::

;   ,    .
```

运算符如下：

```
+   –    *     /     %     &    |     ^     <<     >>    >>>
+=   -=   *=     /=    %=    &=   |=    ^=     <<=   >>=   >>>=
=    ==   !=     <     <=    >     >=
!    ~     &&   ||    ++    --    ?     :    ->
```

整本书中都会见到分隔符，2.4 节会分别介绍每个运算符。

2.3 基本数据类型

Java 支持八种基本数据类型，包括一种布尔类型、一种字符类型、四种整数类型和两种浮点数类型，如表 2-1 所示。四种整数类型和两种浮点数类型的区别在于位数不同，因此能表示的数字范围也不同。

表2-1：Java的基本数据类型

类　型	取　值	默认值	大　小	范　围
`boolean`	`true` 或 `false`	`false`	1 位	NA
`char`	Unicode 字符	`\u0000`	16 位	`\u0000~\uFFFF`
`byte`	有符号的整数	0	8 位	−128~127

（续）

类　型	取　值	默认值	大　小	范　围
`short`	有符号的整数	0	16 位	−32768~32767
`int`	有符号的整数	0	32 位	−2147483648~2147483647
`long`	有符号的整数	0	64 位	−9223372036854775808~9223372036854775807
`float`	IEEE 754 浮点数	0.0	32 位	1.4E−45~3.4028235E+38
`double`	IEEE 754 浮点数	0.0	64 位	4.9E−324~1.7976931348623157E+308

下面几节简要介绍这些基本数据类型。除了基本数据类型之外，Java 还支持称为引用类型的非基本数据类型，2.9 节会介绍。

2.3.1　布尔类型

布尔类型（`boolean`）表示真值，只有两个可选值，表示两种逻辑状态：开或关，是或否，真或假。Java 使用保留字 `true` 和 `false` 表示这两个布尔值。

从其他编程语言，尤其是 JavaScript，转到 Java 的程序员要注意，Java 比其他语言对布尔值的要求严格得多：布尔类型既不是整数类型也不是对象类型，而且不能使用不兼容的值代替布尔类型。也就是说，在 Java 中不能使用下面的简写形式：

```
Object o = new Object();
int i = 1;

if (o) {
  while(i) {
    // ...
  }
}
```

相反，Java 强制要求编写简洁的代码，明确表明想做什么比较：

```
if (o != null) {
  while(i != 0) {
    // ...
  }
}
```

2.3.2　字符类型

字符类型（`char`）表示 Unicode 字符。Java 使用一种稍微独特的方式表示字符：在传给 `javac` 的输入中，标识符使用 UTF-8 编码（一种变长编码方式），但在内部使用定长编码（16 位）表示字符。

不过，开发者一般无需担心这个区别。大多数情况下，只需记住，如果想在 Java 程序中使用字符字面量，只需把字符放在单引号中即可：

```
char c = 'A';
```

当然，字符字面量可以使用任何一个 Unicode 字符，也可以使用 Unicode 转义序列 `\u`。而且，Java 还支持一些其他转义序列，用来表示常用的非打印 ASCII 字符，例如换行符以及转义 Java 中某些有特殊意义的标点符号。例如：

```
char tab = '\t', nul = '\000', aleph = '\u05D0', slash = '\\';
```

表 2-2 列出了可在字符字面量中使用的转义字符。这些字符也可以在字符串字面量中使用，下一节会介绍。

表2-2：Java转义字符

转义序列	字符值
`\b`	退格符
`\t`	水平制表符
`\n`	换行符
`\f`	换页符
`\r`	回车符
`\"`	双引号
`\'`	单引号
`\\`	反斜线
`\xxx`	xxx 编码的 Latin-1 字符，其中 xxx 是八进制数，介于 000 到 377 之间。`\x` 和 `\xx` 两种形式也是合法的，例如 `\0`，但不推荐这么用，因为转义序列只有一个数字，在字符串常量中会导致歧义。这种用法在 `\uxxxx` 中也不鼓励使用
`\uxxxx`	xxxx 编码的 Unicode 字符，其中 xxxx 是四个十六进制数。Unicode 转义序列可以出现在 Java 程序的任意位置，而不只局限于字符和字符串字面量

字符可以转换成整数类型，也可以从整数类型转换而来。字符类型对应的是 16 位整数类型。字符类型与 `byte`、`short`、`int` 和 `long` 不同，没有符号。`Character` 类定义了一些有用的静态方法（static method），用于处理字符，例如 `isDigit()`、`isJavaLetter()`、`isLowerCase()` 和 `toUpperCase()`。

设计 Java 语言和字符类型时考虑到了 Unicode。Unicode 标准一直在发展，每一个 Java 新版本都会使用最新版 Unicode。Java 7 使用的是 Unicode 6.0，Java 8 使用的是 Unicode 6.2。

最近的几版 Unicode 收录了 16 位编码（或叫*码位*，codepoint）无法容纳的字符。这些追加的字符是十分少见的汉字象形文字，占用了 21 位，无法使用单个字符表示，必须使用 `int` 类型表示，或者必须使用“代理对”（surrogate pair）通过两个字符表示。

除非经常使用亚洲语言编写程序，否则很少会遇到这些追加的字符。如果预计要处理无法使用单个字符类型表示的字符，就可以使用 `Character` 和 `String` 等相关类中提供的方法，使用 `int` 类型表示码位，然后再处理文本。

字符串字面量

除了字符类型之外，Java 还有一种用于处理字符串的数据类型。不过，String 类型是类，不是基本类型。因为字符串很常用，所以 Java 提供了一种句法，可以直接在程序中插入字符串。字符串字面量是包含在双引号中的任意文本（字符字面量使用单引号）。例如：

```
"Hello, world"
"'This' is a string!"
```

字符串字面量中可以包含能在字符字面量中使用的任何一个转义序列（参见表 2-2）。如果想在字符串字面量中插入双引号，可以使用 \" 转义序列。String 是引用类型，本章后面的 2.7.4 节还会深入介绍字符串字面量。第 9 章会更详细地介绍在 Java 中处理 String 对象的一些方式。

2.3.3　整数类型

Java 中的整数类型有 byte、short、int 和 long 四种。如表 2-1 所示，这四种类型之间唯一的区别是位数，即能表示的数字范围有所不同。所有整数类型都表示有符号的数字，Java 没有 C 和 C++ 中的 unsigned 关键字。

这四种类型的字面量形式正如你设想的那样，使用十进制数字，前面还可以加上负号。[1] 下面是一些合法的整数字面量：

```
0
1
123
-42000
```

整数字面量还可以使用十六进制、二进制和八进制形式来表示。以 0x 或 0X 开头的字面量是十六进制数，使用字母 A 到 F（或 a 到 f）表示数字的十六进制形式。

整数字面量的二进制形式以 0b 开头，当然，只能使用数字 1 或 0。字面量的二进制形式可能很长，所以经常在字面量中使用下划线。在任何数字字面量中，下划线都会被忽略。下划线纯粹是为了提升字面量的可读性。

Java 还支持使用八进制表示整数字面量，以 0 开头，而且不能使用数字 8 或 9。这种字面量不常用，除非有必要，否则应该避免使用。下面是一些合法的十六进制、二进制和八进制字面量：

```
0xff          // 使用十六进制表示的十进制数255
0377          // 使用八进制表示的十进制数255
0b0010_1111   // 使用二进制表示的十进制数47
0xCAFEBABE    // 用来识别Java类文件的魔法数
```

注 1：严格来说，负号是作用在字面量上的运算符，而不是字面量的一部分。

整数字面量是 32 位 `int` 类型，如果以 `L` 或 `l` 结尾，就表示 64 位 `long` 类型：

```
1234    // int类型
1234L   // long类型
0xffL   // 还是long类型
```

在 Java 中，如果整数运算超出了指定整数类型的范围，不会上溢或下溢，而是直接回绕。例如：

```
byte b1 = 127, b2 = 1;        // byte类型的最大值是127
byte sum = (byte)(b1 + b2);  // 加法运算的结果直接回绕到-128,即byte类型的最小值
```

如果发生了这种情况，Java 编译器和解释器都不会发出任何形式的警告。进行整数运算时，必须确保使用的类型取值范围能满足计算需要。整数除以零，或者计算除以零后得到的余数，都是非法操作，会抛出 `ArithmeticException` 异常。

每一种整数类型都有对应的包装类：`Byte`、`Short`、`Integer` 和 `Long`。这些类都定义了 `MIN_VALUE` 和 `MAX_VALUE` 常量，表示相应的取值范围。而且还定义了一些有用的静态方法，例如 `Byte.parseByte()` 和 `Integer.parseInt()`，作用是把字符串转换成整数。

2.3.4　浮点数类型

在 Java 中，实数使用 `float` 和 `double` 数据类型表示。如表 2-1 所示，`float` 类型是 32 位单精度浮点数，`double` 是 64 位双精度浮点数。这两种类型都符合 IEEE 754-1985 标准。这个标准规定了浮点数的格式和运算方式。

浮点数可以以字面量形式插入 Java 程序，其格式为一些可选的数字，后跟一个小数点和一些数字。下面是几个示例：

```
123.45
0.0
.01
```

浮点数字面量还可以使用指数形式（也叫科学记数法）表示，其格式为一个数后面跟着字母 `e` 或 `E` 和一个数。第二个数表示 10 的次方，是第一个数的乘数。例如：

```
1.2345E02    // 1.2345 * 10^2或123.45
1e-6         // 1 * 10^-6或0.000001
6.02e23      // 阿伏加德罗常数:6.02 * 10^23
```

默认情况下，浮点数是 `double` 类型。若想在程序中插入 `float` 类型的字面量，要在数字后面加上 `f` 或 `F`：

```
double d = 6.02E23;
float f = 6.02e23f;
```

浮点数字面量不能使用十六进制、二进制或八进制表示。

> **浮点数表示的值**
>
> 由于本质上的限制，大多数实数都不能使用有限的位数进行精确表示。因此，要记住，float 和 double 类型都只能表示实际值的近似值。float 类型是 32 位近似值，至少有 6 个有效数字；double 是 64 位近似值，至少有 15 个有效数字。第 9 章会更详细地说明浮点数表示的值。

除了表示普通的数字之外，float 和 double 类型还能表示四个特殊的值：正无穷大、负无穷大、零和 NaN。如果浮点数运算的结果超出了 float 或 double 能表示的范围上限，得到的是无穷大。如果浮点数的运算结果超出了 float 或 double 能表示的范围下限，得到的是零。

Java 的浮点类型区分正零和负零，具体是哪个值取决于从哪个方向出现的下溢。在实际使用中，正零和负零的表现基本一样。最后一种特殊的浮点数 NaN，是“Not-a-Number”的简称，表示“不是数字”。如果浮点数运算不合法，例如 0.0/0.0，得到的就是 NaN。以下几个例子得到的结果就是这些特殊的值：

```
double inf = 1.0/0.0;          // 无穷大
double neginf = -1.0/0.0;      // 负无穷大
double negzero = -1.0/inf;     // 负零
double NaN = 0.0/0.0;          // NaN
```

Java 浮点数类型能处理到无穷大的上溢以及到零的下溢，因此浮点数运算从不抛出异常，就算执行非法运算也没事，例如零除以零，或计算负数的平方根。

float 和 double 基本类型都有对应的类，分别为 Float 和 Double。这两个类都定义了一些有用的常量：MIN_VALUE、MAX_VALUE、NEGATIVE_INFINITY、POSITIVE_INFINITY 和 NaN。

无穷大浮点数的表现和设想的一样，例如，无穷大之间的加减运算得到的还是无穷大。负零的表现几乎和正零一样，而且事实上，相等运算符 == 会告诉你，负零和正零是相等的。区分负零、正零和普通的零有一种方法——把它作为被除数：1.0/0.0 得到的是正无穷大，但是 1.0 除以负零得到的是负无穷大。因为 NaN 不是数字，所以 == 运算符会告诉我们它不等于任何其他数字，甚至包括它自己。若想检查某个 float 或 double 值是否为 NaN，必须使用 Float.isNaN() 或 Double.isNaN() 方法。

2.3.5 基本类型之间的转换

Java 允许整数和浮点数之间相互转换。而且，由于每个字符都对应 Unicode 编码中的一个数字，所以字符与整数和浮点数之间也可以相互转换。其实，在 Java 中，布尔值是唯一一种不能和其他基本类型之间相互转换的基本类型。

类型转换有两种基本方式。把某种类型的值转换成取值范围更广的类型，此时执行的是放大转换（widening conversion）。例如，把 int 字面量赋值给 double 类型的变量和把字符字面量赋值给 int 类型的变量时，Java 会执行放大转换。

另一种方式是缩小转换（narrowing conversion）。把一个值转换成取值范围没那么广的类型时执行的就是缩小转换。缩小转换并不总是安全的，例如把整数 13 转换成 byte 类型是合理的，但把 13 000 转换成 byte 类型就不合理，因为 byte 类型只能介于 −128 和 127 之间。缩小转换可能丢失数据，所以试图缩小转换时 Java 编译器会发出警告，就算转换后的值能落在更窄的取值范围内也会警告：

```
int i = 13;
byte b = i;    // 编译器不允许这么做
```

不过有个例外，如果整数字面量（int 类型）的值落在 byte 和 short 类型的取值范围内，就能把这个字面量赋值给 byte 或 short 类型的变量。

如果需要执行缩小转换，而且确信这么做不会丢失数据或精度，可以使用一种称为“校正”（cast）的语言结构强制 Java 转换。若想执行类型校正，可以在想转换的值前面加一个括号，在括号里写上希望转换成哪种类型。例如：

```
int i = 13;
byte b = (byte) i;   // 把int类型强制转换成byte类型
i = (int) 13.456;    // 把double字面量强制转换成int类型,得到的是13
```

基本类型的校正最常用于把浮点数转换成整数。执行这种转换时，浮点数的小数部分会被直接截掉，即浮点数向零而不是临近的整数舍入。静态方法 Math.round()、Math.floor() 和 Math.ceil() 执行的是另一些舍入方式。

大多数情况下，字符类型的表现都和整数类型类似，所以需要 int 或 long 类型的地方都可以使用字符。不过，还记得吗，字符类型没有符号，所以即便字符和 short 类型都是 16 位，表现上也有差异：

```
short s = (short) 0xffff; // 这些比特表示数字-1
char c = '\uffff';        // 还是这些比特,表示一个Unicode字符
int i1 = s;               // 把short类型转换成int类型,得到的是-1
int i2 = c;               // 把字符转换成int类型,得到的是65535
```

表 2-3 列出了各种基本类型能转换成何种其他类型，以及转换的方式。其中，字母 N 表示无法转换；字母 Y 表示放大转换，由 Java 自动隐式转换；字母 C 表示缩小转换，需要显式校正。

最后，Y* 表示自动执行的放大转换，但在转换过程中最低有效位可能丢失。把 int 或 long 类型转换成浮点类型时可能会出现这种情况，详情参见下表。浮点类型的取值范围比整数类型广，所以 int 或 long 类型都能用 float 或 double 类型来表示。然而，浮点类型

是近似值，所以有效数字不一定总与整数类型一样多（浮点数的详细介绍参见第 9 章）。

表2-3：Java基本类型转换

基本类型	转换为							
	boolean	byte	short	char	int	long	float	double
boolean	-	N	N	N	N	N	N	N
byte	N	-	Y	C	Y	Y	Y	Y
short	N	C	-	C	Y	Y	Y	Y
char	N	C	C	-	Y	Y	Y	Y
int	N	C	C	C	-	Y	Y*	Y
long	N	C	C	C	C	-	Y*	Y*
float	N	C	C	C	C	C	-	Y
double	N	C	C	C	C	C	C	-

2.4 表达式和运算符

到目前为止，我们学习了 Java 程序能处理的基本类型，以及如何在 Java 程序中使用基本类型的*字面量*。还使用了*变量*作为值的符号名称。字面量和变量都是组成 Java 程序的标记。

*表达式*是 Java 程序更高一级的结构。Java 解释器会*求出*表达式的值。最简单的表达式叫*基本表达式*，由字面量和变量组成。例如，下面几个例子都是表达式：

```
1.7     // 一个浮点数字面量
true    // 一个布尔字面量
sum     // 一个变量
```

Java 解释器计算字面量表达式得到的结果是字面量本身；计算变量表达式得到的结果是存储在变量中的值。

基本表达式没什么意思。使用*运算符*把基本表达式连在一起可以组成复杂的表达式。例如，下面的表达式使用赋值运算符把两个基本表达式（一个变量，一个浮点数字面量）连在一起，组成赋值表达式：

```
sum = 1.7
```

不过，运算符不仅能连接基本表达式，也能在任意复杂度的表达式中使用。如下都是合法的表达式：

```
sum = 1 + 2 + 3 * 1.2 + (4 + 8)/3.0
sum/Math.sqrt(3.0 * 1.234)
(int)(sum + 33)
```

2.4.1 运算符概述

一门编程语言能编写什么样的表达式，完全取决于可用的运算符。Java 提供了丰富的运算符，但在有效使用它们之前，要弄清两个重要的概念：优先级和结合性。下面几节详细说明这两个概念和运算符。

1. 优先级

在表 2-4 中，P 列是运算符的优先级。优先级指定运算符执行的顺序。优先级高的运算符在优先级低的运算符之前运算。例如，有如下的表达式：

```
a + b * c
```

乘号的优先级比加号的优先级高，所以 a 和 b 乘以 c 的结果相加，这与小学数学课上学到的一样。运算符的优先级可以理解为运算符和操作数之间绑定的紧密程度，优先级越高，绑定得越紧密。

运算符默认的优先级可以使用括号改变，括号能明确指定运算的顺序。前面那个表达式可以像下面这样重写，先相加再相乘：

```
(a + b) * c
```

Java 采用的默认运算符优先级和 C 语言兼容，C 语言的设计者选定的优先级无需使用括号就能流畅地写出大多数表达式。只有少量的 Java 惯用句法需要使用括号，例如：

```
// 类校正和成员访问结合在一起
((Integer) o).intValue();

// 赋值和比较结合在一起
while((line = in.readLine()) != null) { ... }

// 位运算符和比较结合在一起
if ((flags & (PUBLIC | PROTECTED)) != 0) { ... }
```

2. 结合性

结合性是运算符的一个属性，定义如何计算有歧义的表达式。如果表达式中有多个优先级相同的运算符，结合性尤其重要。

大多数运算符由左至右结合，即从左向右计算。不过，赋值和一元运算符由右至左结合。在表 2-4 中，A 列是运算符或运算符组的结合性，L 表示由左至右，R 表示由右至左。

加号和减号的结合性都是由左至右，所以表达式 a+b-c 从左向右计算，即 (a+b)-c。一元运算符和赋值运算符从右向左计算。例如下面这个复杂的表达式：

```
a = b += c = -~d
```

计算的顺序是：

```
a = (b += (c = -(~d)))
```

和运算符的优先级一样，运算符的结合性也建立了计算表达式的默认顺序。这个默认的顺序可以使用括号改变。然而，Java 选定的默认运算符结合性是为了使用流畅的句法编写表达式，几乎不需要改变。

3. 运算符总结表

表 2-4 总结了 Java 提供的运算符。P 列和 A 列分支表示每类相关运算符的优先级和结合性。这张表可以作为运算符（特别是优先级）的快速参考指南。

表2-4：Java运算符

P	A	运算符	操作数类型	执行的运算
16	L	.	对象，成员	访问对象成员
		[ ]	数组，int	获取数组中的元素
		(args)	方法，参数列表	调用方法
		++, --	变量	后递增，后递减
15	R	++, --	变量	前递增，前递减
		+, -	数字	正号，负号
		~	整数	按位补码
		!	布尔值	逻辑求反
14	R	new	类，参数列表	创建对象
		(type)	类型，任何类型	校正（类型转换）
13	L	*, /, %	数字，数字	乘法，除法，求余数
12	L	+, -	数字，数字	加法，减法
		+	字符串，任何类型	字符串连接
11	L	<<	整数，整数	左移
		>>	整数，整数	右移，高位补符号
		>>>	整数，整数	右移，高位补零
10	L	<, <=	数字，数字	小于，小于或等于
		>, >=	数字，数字	大于，大于或等于
		instanceof	引用类型，类型	类型比较
9	L	==	基本类型，基本类型	等于（值相同）
		!=	基本类型，基本类型	不等于（值不同）
		==	引用类型，引用类型	等于（指向同一个对象）
		!=	引用类型，引用类型	不等于（指向不同的对象）
8	L	&	整数，整数	位与
		&	布尔值，布尔值	逻辑与
7	L	^	整数，整数	位异或
		^	布尔值，布尔值	逻辑异或
6	L	|	整数，整数	位或
		|	布尔值，布尔值	逻辑或

（续）

P	A	运算符	操作数类型	执行的运算
5	L	`&&`	布尔值，布尔值	条件与
4	L	`||`	布尔值，布尔值	条件或
3	R	`? :`	布尔值，任何类型	条件（三元）运算符
2	R	`=`	变量，任何类型	赋值
		`*=`，`/=`，`%=`，`+=`，`-=`，`<<=`，`>>=`，`>>>=`，`&=`，`^=`，`|=`	变量，任何类型	计算后赋值
1	R	`→`	参数列表，方法体	lambda 表达式

4. 操作数的数量和类型

在表 2-4 中，第 4 列是每种运算符能处理的操作数数量和类型。有些运算符只有一个操作数，这种运算符叫一元运算符。例如，一元减号的作用是改变单个数字的符号：

```
-n                  // 一元减号
```

不过，大多数运算符都是二元运算符，有两个操作数。- 运算符其实还有一种用法：

```
a - b            // 减法运算符是二元运算符
```

Java 还定义了一个三元运算符，经常称作条件运算符，就像是表达式中的 `if` 语句。它的三个操作数由问号和冒号分开，第二个和第三个操作数必须能转换成同一种类型：

```
x > y ? x : y  // 三元表达式;计算x和y哪个大
```

除了需要特定数量的操作数之外，每个运算符还需要特定类型的操作数。表 2-4 中的第 4 列是操作数的类型，其中使用的文本需要进一步说明。

- 数字

  整数、浮点数或字符（即除了布尔类型之外的任何一种基本类型）。因为这些类型对应的包装类（例如 `Character`、`Integer` 和 `Double`）能自动拆包（参见 2.9.4 节），所以在这些地方也能使用相应的包装类。

- 整数

  `byte`、`short`、`int`、`long` 或 `char` 类型的值（获取数组元素的运算符 `[ ]` 不能使用 `long` 类型的值）。因为能自动拆包，所以也能使用 `Byte`、`Short`、`Integer`、`Long` 和 `Character` 类型的值。

- 引用类型

  对象或数组。

- 变量

  变量或其他符号名称（例如数组中的元素），只要能赋值就行。

5. 返回类型

就像运算符只能处理特定类型的操作数一样，运算得到的结果也是特定类型的值。对算术运算符、递增和递减、位运算符和位移运算符来说，如果至少有一个操作数是 `double` 类型，返回值就是 `double` 类型；如果至少有一个操作数是 `float` 类型，返回值是 `float` 类型；如果至少有一个操作数是 `long` 类型，返回值是 `long` 类型；除此之外都返回 `int` 类型的值，就算两个操作数都是 `byte`、`short` 或 `char` 类型，也会放大转换成 `int` 类型。

比较、相等性和逻辑运算符始终返回布尔值。各种赋值运算符都返回赋予的值，类型和表达式左边的变量兼容。条件运算符返回第二个或第三个操作数（二者的类型必须相同）。

6. 副作用

每个运算符都会计算一个或多个操作数，得到一个结果。但是，有些运算符除了基本的计算之外还有副作用。如果表达式有副作用，计算时会改变 Java 程序的状态，即再次执行时会得到不同的结果。

例如，`++` 递增运算符的副作用是递增变量中保存的值。表达式 `++a` 会递增变量 `a` 中的值，返回递增后得到的值。如果再次计算这个表达式，会得到不同的值。各种赋值运算符也有副作用。例如，表达式 `a\*=2` 也可以写成 `a=a\*2`，这个表达式的结果是乘于 2 后得到的值，但是有副作用——把计算结果重新赋值给 `a`。

如果调用的方法有副作用，方法调用运算符 `()` 也有副作用。有些方法，例如 `Math.sqrt()`，只是计算后返回一个值，没有任何副作用。可是，一般情况下，方法都有副作用。最后，`new` 运算符有重大的副作用，它会创建一个新对象。

7. 计算的顺序

Java 解释器计算表达式时，会按照表达式中的括号、运算符的优先级和结合性指定的顺序运算。不过，在任何运算之前，解释器会先计算运算符的操作数（`&&`、`||` 和 `?:` 例外，不会总是计算这些运算符的全部操作数）。解释器始终使用从左至右的顺序计算操作数。如果操作数是有副作用的表达式，这种顺序就很重要了。例如下面的代码：

```
int a = 2;
int v = ++a + ++a * ++a;
```

虽然乘法的优先级比加法高，但是会先计算 `+` 运算符的两个操作数。因为这两个操作数都是 `++a`，所以得到的计算的结果分别是 `3` 和 `4`，因此这个表达式计算的是 `3 + 4 * 5`，结果为 `23`。

2.4.2　算术运算符

算术运算符可用于整数、浮点数和字符（即除了布尔类型之外的所有基本类型）。如果其中有个操作数是浮点数，就按浮点算术运算；否则，按整数算术运算。这一点很重要，因

为整数算术和浮点算术是有区别的，例如除法的运算方式，以及上溢和下溢的处理方式。算术运算符如下。

- 加法（+）

  + 号计算两个数之和。稍后会看到，+ 号还能连接字符串。如果 + 号的操作数中有一个是字符串，另一个也会转换成字符串。如果想把加法和连接放在一起使用，一定要使用括号。例如：

  ```
  System.out.println("Total: " + 3 + 4);    // 打印"Total: 34",不是7!
  ```

- 减法（-）

  - 号当成二元运算符使用时，计算第一个操作数减去第二个操作数得到的结果。例如，7-3 的结果是 4。- 号也可执行一元取负操作。

- 乘法（*）

  * 号计算两个操作数的乘积。例如，7*3 的结果是 21。

- 除法（/）

  / 号用第一个操作数除以第二个操作数。如果两个操作数都是整数，结果也是整数，丢掉余数。如果有一个操作数是浮点数，结果就是浮点数。两个整数相除时，如果除数是零，抛出 `ArithmeticException` 异常。不过，对浮点数计算来说，如果除以零，得到的是无穷大或 NaN：

  ```
  7/3         // 计算结果为2
  7/3.0f      // 计算结果为2.333333f
  7/0         // 抛出ArithmeticException异常
  7/0.0       // 计算结果为正无穷大
  0.0/0.0     // 计算结果为NaN
  ```

- 求模（%）

  % 运算符计算第一个操作数和第二个操作数的模数，即返回第一个操作数除去第二个操作数的整倍数之后剩下的余数。例如，7%3 的结果是 1。结果的符号和第一个操作数的符号一样。虽然求模运算符的操作数一般是整数，但也可以使用浮点数。例如，4.3%2.1 的结果是 0.1。如果操作数是整数，计算零的模数会抛出 `ArithmeticException` 异常。如果操作数是浮点数，计算 0.0 的模数得到的结果是 NaN；计算无穷大和任何数的模数得到的结果也是 NaN。

- 负号（-）

  如果把 - 号当成一元运算符使用，即放在单个操作数之前，执行的是一元取负运算。也就是说，会把正数转换成对应的负数，或把负数转换成对应的正数。

2.4.3 字符串连接运算符

+ 号（以及相关的 += 运算符）除了能计算数字之和以外，还能连接字符串。如果 + 号的两个操作数中有一个是字符串，另一个操作数也会转换成字符串。例如：

```
// 打印“Quotient: 2.3333333”
System.out.println("Quotient: " + 7/3.0f);
```

因此，如果加法和字符串连接结合在一起使用，要把加法表达式放在括号中。如果不这么做，加号会被理解成连接运算符。

Java 解释器原生支持把所有基本类型转换成字符串。对象转换成字符串时，调用的是对象的 toString() 方法。有些类自定义了 toString() 方法，所以这些类的对象可以使用这种方式轻易地转换成字符串。数组转换成字符串时会调用原生的 toString() 方法，不过可惜，这个方法没有为数组的内容提供有用的字符串形式。

2.4.4 递增和递减运算符

++ 运算符把它的单个操作数增加 1，这个操作数必须是变量、数组中的元素或对象的字段。这个运算符的行为取决于它相对于操作数的位置。放在操作数之前，是前递增运算符，递增操作数的值，并返回递增后的值。放在操作数之后，是后递增运算符，递增操作数的值，但返回递增前的值。

例如，下面的代码把 i 和 j 的值都设为 2：

```
i = 1;
j = ++i;
```

但是，下面的代码把 i 的值设为 2，j 的值设为 1：

```
i = 1;
j = i++;
```

类似地，-- 运算符把它的单个数字操作数减小 1，这个操作数必须是变量、数组中的元素或对象的字段。和 ++ 运算符一样，-- 的行为也取决于它相对于操作数的位置。放在操作数之前，递减操作数的值，并返回递减后的值。放在操作数之后，递减操作数的值，但返回递减前的值。

表达式 x++ 和 x-- 分别等效于 x=x+1 和 x=x-1，不过使用递增和递减运算符时，只会计算一次 x 的值。如果 x 是有副作用的表达式，情况就大不相同了。例如，下面两个表达式不等效：

```
a[i++]++;              // 递增数组中的一个元素
// 把数组中的一个元素增加1,然后把新值存储在另一个元素中
a[i++] = a[i++] + 1;
```

这些运算符，不管放在前面还是后面，最常用来递增或递减控制循环的计数器。

2.4.5 比较运算符

比较运算符包括测试两个值是否相等的相等运算符和测试有序类型（数字和字符）数据之间大小关系的关系运算符。这两种运算符计算的结果都是布尔值，因此一般用于 `if` 语句、`while` 和 `for` 循环，作为分支和循环的判定条件。例如：

```
if (o != null) ...;         // 不等运算符
while(i < a.length) ...;    // 小于运算符
```

Java 提供了下述相等运算符。

- 等于（==）

  如果 == 运算符的两个操作数相等，计算结果为 `true`；否则计算结果为 `false`。如果操作数是基本类型，这个运算符测试两个操作数的值是否一样。如果操作数是引用类型，这个运算符测试两个操作数是否指向同一个对象或数组。尤其要注意，这个运算符不能测试两个字符串是否相等。

  如果使用 == 比较两个数字或字符，而且两个操作数的类型不同，在比较之前会把取值范围窄的操作数转换成取值范围宽的操作数类型。例如，比较 `short` 类型的值和 `float` 类型的值时，在比较之前会先把 `short` 类型的值转换成 `float` 类型。对浮点数来说，特殊的负零和普通的正零相等；特殊的 NaN 和任何数，包括 NaN 自己，都不相等。如果想测试浮点数是否为 NaN，要使用 `Float.isNan()` 或 `Double.isNan()` 方法。

- 不等于（!=）

  != 运算符完全是 == 运算符的反运算。如果两个基本类型操作数的值不同，或者两个引用类型操作数指向不同的对象或数组，!= 运算符的计算结果为 `true`；否则，计算结果为 `false`。

关系运算符可用于数字和字符，但不能用于布尔值、对象和数组，因为这些类型无序。Java 提供了下述关系运算符。

- 小于（<）

  如果第一个操作数小于第二个操作数，计算结果为 `true`。

- 小于或等于（<=）

  如果第一个操作数小于或等于第二个操作数，计算结果为 `true`。

- 大于（>）

  如果第一个操作数大于第二个操作数，计算结果为 `true`。

- 大于或等于（>=）

  如果第一个操作数大于或等于第二个操作数，计算结果为 true。

2.4.6 逻辑运算符

如前所示，比较运算符比较两个操作数，计算结果为布尔值，经常用在分支和循环语句中。为了让分支和循环的条件判断更有趣，可以使用逻辑运算符把多个比较表达式合并成一个更复杂的表达式。逻辑运算符的操作数必须是布尔值，而且计算结果也是布尔值。逻辑运算符如下。

- 条件与（&&）

  这个运算符对操作数执行逻辑与运算。仅当两个操作数都是 true 时才返回 true；如果有一个或两个操作数都是 false，计算结果为 false。例如：

  ```
  if (x < 10 && y > 3) ... // 如果两个比较表达式的结果都是true
  ```

  这个运算符（以及除了一元运算符 ! 之外的所有逻辑运算符）的优先级没有比较运算符高，因此完全可以编写类似上面的代码。不过，有些程序员倾向于使用括号，明确表明计算的顺序：

  ```
  if ((x < 10) && (y > 3))...
  ```

  你觉得哪种写法更易读就用哪种。

  这个运算符之所以叫条件与，是因为它会视情况决定是否计算第二个操作数。如果第一个操作数的结算结果为 false，不管第二个操作数的计算结果是什么，这个表达式的计算结果都是 false。因此，为了提高效率，Java 解释器会走捷径，跳过第二个操作数。因为不一定会计算第二个操作数，所以使用这个运算符时，如果表达式有副作用，一定要注意。不过，因为有这种特性，可以使用这个运算符编写如下的 Java 表达式：

  ```
  if (data != null && i < data.length && data[i] != -1)
      ...
  ```

  如果第一个和第二个比较表达式的计算结果为 false，第二个和第三个比较表达式会导致错误。幸好，我们无需为此担心，因为 && 运算符会视情况决定是否执行后面的表达式。

- 条件或（||）

  这个运算符在两个布尔值操作数上执行逻辑或运算。如果其中一个或两个都是 true，计算结果为 true；如果两个操作数都是 false，计算结果为 false。和 && 运算符一样，|| 并不总会计算第二个操作数。如果第一个操作数的计算结果为 true，不管第二个操作数的计算结果是什么，表达式的计算结果都是 true。因此，遇到这种情况时，|| 运

算符会跳过第二个操作数。

- 逻辑非（!）

  这个运算符改变操作数的布尔值。如果应用于 true，计算结果为 false；如果应用于 false，计算结果为 true。在下面这种表达式中很有用：

  ```
  if (!found) ...          // found是其他地方定义的布尔值
  while (!c.isEmpty()) ... // isEmpty()方法返回布尔值
  ```

  ! 是一元运算符，优先级高，经常必须使用括号：

  ```
  if (!(x > y && y > z))
  ```

- 逻辑与（&）

  如果操作数是布尔值，& 运算符的行为和 && 运算符类似，但是不管第一个操作数的计算结果如何，总会计算第二个操作数。不过，这个运算符几乎都用作位运算符，处理整数操作数。很多 Java 程序员都认为使用这个运算符处理布尔值操作数是不合法的 Java 代码。

- 逻辑或（|）

  这个运算符在两个布尔值操作数上执行逻辑或运算，和 || 运算符类似，但是就算第一个操作数的计算结果为 true，也会计算第二个操作数。| 运算符几乎都用作位运算符，处理整数操作数，很少用来处理布尔值操作数。

- 逻辑异或（^）

  如果操作数是布尔值，这个运算符的计算结果是两个操作数的异或。如果两个操作数中只有一个是 true，计算结果才是 true。也就是说，如果两个操作数都是 true 或 false，计算结果为 false。这个运算符与 && 和 || 不同，始终会计算两个操作数。^ 运算符更常用作位运算符，处理整数操作数。如果操作数是布尔值，这个运算符等效于 != 运算符。

2.4.7　位运算符和位移运算符

位运算符和位移运算符是低层运算符，处理组成整数的单个位。现代 Java 程序很少使用位运算符，除非处理低层操作（例如网络编程）。这两种运算符用于测试和设定整数中的单个标志位。若想理解这些运算符的行为，必须先理解二进制数以及用于表示负整数的二进制补码方式。

这些运算符的操作数不能是浮点数、布尔值、数组或对象。如果操作数是布尔值，&、| 和 \^ 运算符执行的是其他运算，前一节已经讲过。

如果位运算符的操作数中有一个是 long 类型，结果就是 long 类型。除此之外，结果都是 int 类型。如果位移运算符左边的操作数是 long 类型，结果为 long 类型；否则，结果是

int 类型。位运算符和位移运算符如下。

- *按位补码*（~）

  一元运算符 ~ 是按位补码运算符，或叫位或运算符。它把单个操作数的每一位反相，1 变成 0，0 变成 1。例如：

```
byte b = ~12;            // ~00001100 = => 11110011或十进制数-13
flags = flags & ~f;      // 把标志集合flags中的f标志清除
```

- *位与*（&）

  这个运算符在两个整数操作数的每一位上执行逻辑与运算，合并这两个操作数。只有两个操作数的同一位都为 1 时，结果中对应的位才是 1。例如：

```
10 & 7                   // 00001010 & 00000111 ==> 00000010或2
if ((flags & f) != 0)    // 测试是否设定了f标志
```

  前面已经说过，如果操作数是布尔值，& 是不常使用的逻辑与运算符。

- *位或*（|）

  这个运算符在两个整数操作数的每一位上执行逻辑或运算，合并这两个操作数。如果两个操作数的同一位中有一个或两个都是 1，结果中对应的位是 1；如果两个操作数的同一位都是 0，结果中对应的位是 0。例如：

```
10 | 7                  // 00001010 | 00000111 ==> 00001111或15
flags = flags | f;      // 设定f标志
```

  前面已经说过，如果操作数是布尔值，| 是不常使用的逻辑或运算符。

- *位异或*（^）

  这个运算符在两个整数操作数的每一位上执行逻辑异或运算，合并这两个操作数。如果两个操作数的同一位值不同，结果中对应的位是 1；如果两个操作数的同一位都是 1 或都是 0，结果中对应的位是 0。例如：

```
10 ^ 7     // 00001010 ^ 00000111 ==> 00001101或13
```

  如果操作数是布尔值，^ 是很少使用的逻辑异或运算符。

- *左移*（<<）

  << 运算符把左侧操作数的每一位向左移动右侧操作数指定的位数。左侧操作数的高位被丢掉，右边缺少的位补零。整数向左移 n 位，相当于乘于 $2n$。例如：如果左侧操作数是 long 类型，右侧操作数应该介于 0 和 63 之间。

```
10 << 1    // 00001010 << 1 = 00010100 = 20 = 10*2
7 << 3     // 00000111 << 3 = 00111000 = 56 = 7*8
-1 << 2    // 0xFFFFFFFF << 2 = 0xFFFFFFFC = -4 = -1*4
```

  如果左侧操作数是 int 类型，右侧操作数应该介于 0 和 31 之间。

- *带符号右移（>>）*

  >> 运算符把左侧操作数的每一位向右移动右侧操作数指定的位数。左侧操作符的低位被移除，移入的高位和原来的最高位一样。也就是说，如果左侧操作数是正数，移入的高位是 0；如果左侧操作数是负数，移入的高位是 1。这种技术叫*高位补符号*，作用是保留左侧操作数的符号。例如：

  ```
  10 >> 1     // 00001010 >> 1 = 00000101 = 5 = 10/2
  27 >> 3     // 00011011 >> 3 = 00000011 = 3 = 27/8
  -50 >> 2    // 11001110 >> 2 = 11110011 = -13 != -50/4
  ```

  如果左侧操作数是正数，右侧操作数是 n，>> 运算符的计算结果相当于整数除以 2^n。

- *不带符号右移（>>>）*

  这个运算符和 >> 类似，但是不管左侧操作数的符号是什么，高位总是移入 0。这种技术叫*高位补零*。左侧操作数是无符号的数字时才适用这个运算符（可是 Java 的整数类型都带符号）。下面是一些例子：

  ```
  0xff >>> 4     // 11111111 >>> 4 = 00001111 = 15 = 255/16
  -50 >>> 2     // 0xFFFFFFCE >>> 2 = 0x3FFFFFF3 = 1073741811
  ```

2.4.8 赋值运算符

赋值运算符把值存储在某种变量中或赋予某种变量。左侧操作数必须是适当的局部变量、数组元素或对象字段。右侧操作数可以是与变量兼容的任何类型。赋值表达式的计算结果是赋予变量的值。不过，更重要的是，赋值表达式的副作用是执行赋值操作。和其他二元运算符不同的是，赋值运算符是右结合的，也就是说，赋值表达式 `a=b=c` 从右向左执行，即 `a=(b=c)`。

基本的赋值运算符是 `=`。别把它和相等运算符 `==` 搞混了。为了区别这两个运算符，我们建议你把 `=` 读作“被赋值为”。

除了这个简单的赋值运算符之外，Java 还定义了另外 11 个运算符，其中 5 个与算术运算符一起使用，6 个与位运算符和位移运算符一起使用。例如，`+=` 运算符先读取左侧变量的值，再和右侧操作数相加。这种表达式的副作用是把两数之和赋值给左侧变量，返回值也是两数之和。因此，表达式 `x+=2` 几乎和 `x=x+2` 一样。这两种表达式之间的区别是，`+=` 运算符只会计算一次左侧操作数。如果左侧操作数有副作用，这个区别就体现出来了。如下两个表达式并不等效：

```
a[i++] += 2;
a[i++] = a[i++] + 2;
```

组合赋值运算符的一般格式为：

```
var op= value
```

（如果 var 没有副作用）等效于：

```
var = var op value
```

可用的组合赋值运算符有：

```
+=    -=    *=    /=    %=  // 算术运算符加赋值运算符
&=    |=    ^=              // 位运算符加赋值运算符
<<=   >>=   >>>=            // 位移运算符加赋值运算符
```

其中，最常用的运算符是 += 和 -=，不过处理布尔值标志时，&= 和 |= 也有用。例如：

```
i += 2;          // 循环计数器增加2
c -= 5;          // 计数器减小5
flags |= f;      // 在一组整数标志flags中设定f标志
flags &= ~f;     // 在一组整数标志flags中清除f标志
```

2.4.9　条件运算符

条件运算符 ?: 是有点晦涩的三元运算符（有三个操作数），从 C 语言继承而来，可以在一个表达式中嵌入条件判断。这个运算符可以看成是 if/else 语句的运算符版。条件运算符的第一个和第二个操作数使用问号（?）分开，第二个和第三个操作数使用冒号（:）分开。第一个操作数的计算结果必须是布尔值。第二个和第三个操作数可以是任意类型，但要能转换成同一类型。

条件运算符先计算第一个操作数，如果结果为 true，就计算第二个操作数，并把结果当成表达式的返回值；如果第一个操作数的计算结果为 false，条件运算符会计算并返回第三个操作数。条件运算符绝不会同时计算第二个和第三个操作数，所以使用有副作用的表达式时要小心。这个运算符的使用示例如下：

```
int max = (x > y) ? x : y;
String name = (name != null) ? name : "unknown";
```

注意，?: 运算符的优先级只比赋值运算符高，比其他运算符都低，所以一般不用把操作数放在括号里。不过，很多程序员觉得，把第一个操作数放在括号里，条件表达式更易读。的确，毕竟 if 语句的条件表达式都放在括号里。

2.4.10　instanceof操作符

instanceof 操作符与对象和 Java 的类型系统联系紧密。如果你是初次接触 Java，建议你跳过这一节，等你对 Java 的对象有充足了解后再看。

instanceof 操作符的左侧操作数是对象或数组，右侧操作数是引用类型的名称。如果对象

或数组是指定类型的实例，计算结果为 true；否则，计算结果为 false。如果左侧操作数是 null，instanceof 操作符的计算结果始终为 false。如果 instanceof 表达式的计算结果为 true，意味着可以放心校正并把左侧操作数赋值给类型为右侧操作数的变量。

instanceof 操作符只能用于引用类型和对象，不能用于基本类型和值。instanceof 操作符的使用示例如下：

```
// true:所有字符串都是String类的实例
"string" instanceof String
// true:字符串也是Object类的实例
"" instanceof Object
// false:null不是任何类的实例
null instanceof String

Object o = new int[] {1,2,3};
o instanceof int[]   // true:这个数组是int数组
o instanceof byte[]  // false:这个数组不是byte数组
o instanceof Object  // true:所有数组都是Object类的实例

// 使用instanceof确保能放心校正对象
if (object instanceof Point) {
   Point p = (Point) object;
}
```

2.4.11 特殊运算符

Java 有六种语言结构，有时当成运算符，有时只当成基本句法的一部分。表 2-4 也列出了这些“运算符”，以便说明相对于其他真正运算符的优先级。本书其他地方会详细介绍这些语言结构的用法，不过这里要简要说明一下，以便在代码示例中能识别它们。

- *访问对象成员*（.）

  对象由一些数据和处理这些数据的方法组成。对象的数据字段和方法称为这个对象的成员。点号运算符（.）用来访问这些成员。如果 o 是一个表达式，而且计算结果为对象引用，f 是这个对象的字段名称，那么，o.f 的计算结果是字段 f 中的值。如果 m 是一个方法的名称，那么，o.m 指向这个方法，而且能使用后面介绍的 () 运算符调用。

- *访问数组中的元素*（[]）

  数组是由编号的值组成的列表。数组中的每个元素都能使用各自的编号（或叫*索引*）引用。[ ] 运算符能指向数组中的单个元素。如果 a 是一个数组，i 是能计算为 int 类型的表达式，那么，a[i] 指向 a 中的一个元素。这个运算符不像其他处理整数的运算符，它强制要求数组的索引必须是 int 类型或者取值范围更窄的类型。

- *调用方法*（()）

  方法是一些有名称的 Java 代码，在这个名称的后面加上括号，并在括号中放零个或多个以逗号分隔的表达式，可以运行（或叫*调用*）方法。括号中的表达式计算得到的值是

方法的参数。方法会处理这些参数，有时还会返回一个值，这个值是方法调用表达式的返回值。如果 o.m 是一个没有参数的方法，那么这个方法可以使用 o.m() 调用。假设这个方法有三个参数，那么可以使用表达式 o.m(x,y,z) 调用。Java 解释器调用方法之前，会先计算传入的参数。这些表达式始终从左至右计算（如果参数有副作用，就能体现顺序的重要性）。

- lambda表达式（->）

  lambda 表达式是一些匿名的 Java 可执行代码，其实就是方法的主体，由方法的参数列表（零个或多个以逗号分隔的表达式，放在括号中）、lambda 箭头运算符和一段 Java 代码组成。如果代码段只有一个语句，可以省略标识块边界常用的花括号。

- 创建对象（new）

  在 Java 中，对象和数组使用 new 运算符创建。运算符后面跟着想创建的对象类型，括号中还可以指定一些传给对象构造方法的参数。构造方法是一种特殊的代码块，用于实例化新建的对象。创建对象的句法和调用方法的句法类似。例如：

```
new ArrayList();
new Point(1,2)
```

- 类型转换或校正（()）

  前面已经介绍过，括号还可以当成执行缩小类型转换（或叫校正）的运算符。这个运算符的第一个操作数是想转换的类型，放在括号里；第二个操作数是要转换的值，跟在括号后面。例如：

```
(byte) 28          // 把整数字面量校正成byte类型
(int) (x + 3.14f)  // 把浮点数之和校正成整数
(String)h.get(k)   // 把泛型对象校正成字符串
```

2.5 语句

语句是 Java 语言中可执行代码的基本单位，表达程序员的某个意图。和表达式不同，Java 语句没有返回值。语句一般包含表达式和运算符（尤其是赋值运算符），执行的目的往往是为了它们的副作用。

Java 定义的很多语句是流程控制语句，例如条件语句和循环语句，它们通过合理的方式改变默认的线性执行顺序。表 2-5 总结了 Java 定义的语句。

表2-5：Java语句

语　句	作　用	句　法
表达式	副作用	var = expr; expr++; method(); new Type();
复合语句	语句组	{ statements }
空语句	无作用	;

（续）

语　句	作　用	句　法
标注	为语句命名	`label: statement`
变量	声明变量	`[final] type name[=value][,name[=value]] ...;`
if	条件判断	`if (expr) statement[ else statement]`
switch	条件判断	`switch (expr) { [ case expr : statements ] ... [ default: statements ] }`
while	循环	`while (expr) statement`
do	循环	`do statement while (expr);`
for	简单循环	`for (init; test; increment) statement`
遍历	迭代集合	`for (variable : iterable) statement`
break	退出块	`break [label];`
continue	重新开始循环	`continue [label];`
return	结束方法	`return [expr];`
synchronized	临界区	`synchronized (expr) {statements}`
throw	抛出异常	`throw expr;`
try	处理异常	`try {statements}[ catch (type name) { statements } ] ... [ finally { statements } ]`
assert	验证不变式	`assert invariant[ :error];`

2.5.1　表达式语句

本章前面已经说过，某些 Java 表达式有副作用。也就是说，这些表达式不仅能计算得到的一个值，还能以某种方式改变程序的状态。只要表达式有副作用，在表达式后面加上分号就能作为语句使用。合法的表达式语句有赋值、递增和递减、方法调用以及对象创建。例如：

```
a = 1;                            // 赋值
x *= 2;                           // 带运算的赋值
i++;                              // 后递增
--c;                              // 前递减
System.out.println("statement");  // 方法调用
```

2.5.2　复合语句

*复合语句*是一些放在花括号里的语句，语句的数量和类型不限。Java 句法规定可以使用语句的地方都可以使用复合语句：

```
for(int i = 0; i < 10; i++) {
   a[i]++;           // 这个循环体是一个复合语句
   b[i]--;           // 包括两个表达式语句
}                    // 放在花括号里
```

2.5.3　空语句

在 Java 中，*空语句*使用一个分号表示。空语句什么也不做，不过这种句法偶尔有用。例

如，在 `for` 循环中可以使用空语句表明循环体为空：

```
for(int i = 0; i < 10; a[i++]++)  // 递增数组元素
     /* empty */;                 // 循环体是空语句
```

2.5.4 标注语句

*标注语句*就是有名称的语句。命名方法是，在语句前加上一个标识符和一个冒号。`break` 和 `continue` 语句会用到标注。例如：

```
rowLoop: for(int r = 0; r < rows.length; r++) {        // 一个标注循环
   colLoop: for(int c = 0; c < columns.length; c++) {  // 另一个
     break rowLoop;                                    // 使用标注
   }
}
```

2.5.5 局部变量声明语句

*局部变量*经常直接称为变量，是值存储位置的符号名称，在方法和复合语句中定义。所有变量在使用之前必须先声明，声明变量的方法是使用声明语句。Java 是静态类型语言，声明变量时要指定变量的类型，而且只有这种类型的值才能存储在这个变量中。

变量声明语句最简单的形式只需指定变量的类型和名称：

```
int counter;
String s;
```

声明变量时还可以包含一个*初始化表达式*，用于指定变量的初始值。例如：

```
int i = 0;
String s = readLine();
int[] data = {x+1, x+2, x+3}; // 稍后会介绍数组初始化表达式
```

Java 编译器不允许使用未初始化的局部变量，所以，方便起见，通常会在一个语句中同时声明和初始化变量。初始化表达式不必是编译器能计算得到结果的字面量或常量表达式，也可以是程序运行时能计算出结果的任意复杂表达式。

一个变量声明语句可以声明和初始化多个变量，但是所有变量必须是同一类型。变量名称和可选的初始化表达式使用逗号分隔：

```
int i, j, k;
float x = 1.0, y = 1.0;
String question = "Really Quit?", response;
```

变量声明语句可以以 `final` 关键字开头。这个修饰符表明，为变量指定初始值之后，其值就不能改变了：

```
final String greeting = getLocalLanguageGreeting();
```

后文，尤其是讨论不可变编程风格时，还会说明 final 关键字。

C 语言程序员要注意，在 Java 代码的任何地方都能使用变量声明语句，而不局限于只能在方法和代码块的开头使用。稍后会介绍，局部变量声明还可以集成到 for 循环的初始化部分里。

局部变量只能在其定义所在的方法和代码块中使用，这叫变量的作用域或词法作用域。

```
void method() {            // 定义一个方法
   int i = 0;              // 声明变量i
   while (i < 10) {        // 在这个作用域里可以使用i
     int j = 0;            // 声明变量j;j的作用域从这里开始
     i++;                  // 在这个作用域里可以使用i;递增i
   }                       // 在这个作用域里不能使用j了
   System.out.println(i); // 在这个作用域里仍能使用i
}                          // i的作用域在这结束
```

2.5.6 if/else语句

if 语句是基本的控制语句，允许 Java 作出判断，或者更准确地说，根据条件决定执行哪些语句。if 语句有关联的表达式和语句，如果表达式的计算结果为 true，解释器会执行关联的语句；如果表达式的计算结果为 false，解释器会跳过关联的语句。

> Java 允许在关联的表达式中使用包装类型 Boolean 代替基本类型 boolean。此时，包装对象会自动拆包。

下面是一个 if 语句示例：

```
if (username == null)         // 如果username的值是null
   username = "John Doe";     // 使用默认值
```

虽然括号看起来不重要，但却是 if 语句句法不可缺少的一部分。前面说过，花括号中的语句块本身也是语句，所以 if 语句还可以写成这样：

```
if ((address == null) || (address.equals(""))) {
   address = "[undefined]";
   System.out.println("WARNING: no address specified.");
}
```

if 语句可以包含一个可选的 else 关键字，并在后面跟着另一个语句。在这种形式中，如果表达式的计算结果为 true，会执行第一个语句，否则执行第二个语句。例如：

```
if (username != null)
   System.out.println("Hello " + username);
else {
   username = askQuestion("What is your name?");
   System.out.println("Hello " + username + ". Welcome!");
}
```

嵌套使用 if/else 语句时要注意，必须确保 else 子句和正确的 if 语句匹配。例如下面的代码：

```
if (i == j)
   if (j == k)
     System.out.println("i equals k");
else
   System.out.println("i doesn't equal j");    // 错误！！
```

在这个例子中，根据句法，内层 if 语句是外层 if 语句的单个语句。但是，（除了缩进给出的提示）else 子句和哪个 if 语句匹配并不明确。而且，这个例子的缩进提示也是错的。规则是这样的，else 子句和最近的 if 语句关联。正确缩进后的代码如下：

```
if (i == j)
   if (j == k)
     System.out.println("i equals k");
   else
     System.out.println("i doesn't equal j");    // 错误！！
```

这是合法的代码，但显然没有清楚表明程序员的意图。使用嵌套 if 语句时，应该使用花括号，让代码更易读。下面是这个示例更好的编写方式：

```
if (i == j) {
   if (j == k)
     System.out.println("i equals k");
}
else {
   System.out.println("i doesn't equal j");
}
```

else if子句

if/else 语句适用于测试一个条件，并在两个语句或代码块中选择一个执行。那么需要在多个代码块中选择时怎么办呢？这种情况一般使用 else if 子句。这其实不是新句法，而是标准 if/else 语句的惯用句法。用法如下：

```
if (n == 1) {
    // 执行代码块#1
}
else if (n == 2) {
    // 执行代码块#2
}
else if ( n== 3) {
    // 执行代码块#3
```

```
}
else {
    // 如果前面的条件判断都失败,执行代码块#4
}
```

这段代码没什么特别，只是一系列 if 语句，其中各 if 语句是前一个语句 else 子句的一部分。较之完全使用嵌套的形式（如下所示），更推荐使用 else if。

```
if (n == 1) {
   // 执行代码块#1
}
else {
   if (n== 2) {
     // 执行代码块#2
   }
   else {
     if (n == 3) {
       // 执行代码块#3
     }
     else {
       // 如果前面的条件判断都失败,执行代码块#4
     }
   }
}
```

2.5.7 switch语句

if 语句在程序的执行过程中创建一个分支。如前一节所述，可以使用多个 if 语句创建多个分支。但这么做并不总是最好的方式，尤其是所有分支都判断同一个变量的值时，在多个 if 语句中重复检查这个变量的值效率不高。

更好的方式是使用从 C 语言继承而来的 switch 语句。虽然这种语句的句法没有 Java 中其他语句优雅，但是鉴于它的实用性，还是值得使用。

> switch 语句以一个表达式开始，这个表达式的返回值是 int、short、char、byte（或这四个类型的包装类型）、String 或枚举类型（详细介绍参见第 4 章）。

这个表达式后面跟着一段放在花括号里的代码，这段代码中有多个入口点，对应于表达式各个可能的返回值。例如，下面的 switch 语句等效于前一节的多个 if 和 else/if 语句：

```
switch(n) {
   case 1:              // 如果n == 1,从这开始
     // 执行代码块#1
     break;             // 在这停止
   case 2:              // 如果n == 2,从这开始
     // 执行代码块#2
```

```
    break;              // 在这停止
  case 3:               // 如果n == 3,从这开始
    // 执行代码块#3
    break;              // 在这停止
  default:              // 如果前面的条件判断都失败了……
    // 执行代码块#4
    break;              // 在这停止
}
```

从这个示例可以看出，switch 语句中的各入口点有两种形式：一种使用关键字 case 标注，后面跟着一个整数和一个冒号；另一种使用特殊的关键字 default 标注，后面跟着一个冒号。解释器执行 switch 语句时，先计算括号中表达式的值，然后查找有没有匹配这个值的 case 标注。如果有，解释器就从这个 case 标注后的代码块中第一个语句开始执行；如果没有，解释器从特殊的 default 标注后的代码块中第一个语句开始执行；如果没有 default 标注，解释器会跳过整个 switch 语句主体。

注意，在前面的代码中每个 case 子句末尾都有 break 关键字。本章后面会介绍 break 语句，这里，它的作用是让解释器退出 switch 语句的主体。switch 语句中的 case 子句只用来指定需要执行的代码起始点，各 case 子句后的代码块不是相互独立的，没有任何隐式的结束点。因此，必须使用 break 或相关的语句明确指定各 case 子句在哪里结束。如果没有 break 语句，switch 语句会从匹配的 case 标注后第一个语句开始执行，一直到代码块结束为止。极少数的情况下会这样编写代码，从一个 case 标注执行到下一个 case 标注；99% 的情况下都要在每个 case 和 default 子句中加上一个语句，结束执行 switch 语句。一般情况下使用 break 语句，不过 return 和 throw 语句也行。

switch 语句可以使用多个 case 子句标注同一个希望执行的语句。例如下面这个方法中的 switch 语句：

```
boolean parseYesOrNoResponse(char response) {
  switch(response) {
    case 'y':
    case 'Y': return true;
    case 'n':
    case 'N': return false;
    default:
      throw new IllegalArgumentException("Response must be Y or N");
  }
}
```

switch 语句和 case 标注有些重要的限制。首先，switch 语句关联的表达式必须是适当的类型，可以是 byte、char、short、int（及这四种类型的包装类型）、枚举类型或 String 类型，不支持浮点数和布尔类型，虽然 long 也是整数类型，但也不能使用。其次，各 case 标注关联的值必须是编译器能计算的常量或常量表达式。case 标注不能包含运行时表达式，例如变量或方法调用。再者，case 标注中的值必须在 switch 表达式返回值对应数据类型的取值范围内。最后，不能有两个或多个 case 标注使用同一个值，而且 default 标

注不能超过一个。

2.5.8 while语句

while 语句是一种基本语句，目的是让 Java 执行重复的操作。换言之，while 语句是 Java 的主要循环结构之一。句法如下：

```
while (expression)
   statement
```

while 语句先计算 expression 的值，计算结果必须是布尔值。如果计算结果为 false，解释器跳过循环中的 statement，执行程序中的下一个语句。如果计算结果为 true，解释器执行组成循环主体的 statement，然后再次计算 expression 的值。如果计算结果为 false，解释器执行程序中的下一个语句；否则，再次执行 statement。只要 expression 的计算结果为 true，就会一直循环下去，while 语句结束后（即 expression 的计算结果为 false）解释器才会执行下一个语句。

下面是一个 while 循环示例，打印数字 0 到 9：

```
int count = 0;
while (count < 10) {
   System.out.println(count);
   count++;
}
```

可以看出，在这个示例中，变量 count 的起始值是 0，循环主体每执行一次，count 的值就会增加 1。循环执行 10 次后，表达式的计算结果变成 false（即 count 的值不再小于 10），此时 while 语句结束，Java 解释器继续执行程序中的下一个语句。大多数循环都有一个计数器变量，例如这个例子中的 count。循环计数器变量的名称经常使用 i、j 和 k，不过你应该使用意义更明确的名字，以便代码更易理解。

2.5.9 do语句

do 循环和 while 循环很像，不过循环表达式不在循环开头，而在循环末尾测试。也就是说，循环主体至少会执行一次。do 循环的句法如下：

```
do
   statement
while (expression);
```

注意，do 循环和更普通的 while 循环有几个不同点。首先，do 循环既需要使用关键字 do 标记循环的开头，也要使用关键字 while 标记循环的结尾，以及引入循环条件。其次，与 while 循环不同的是，do 循环的结尾要使用分号。这是因为 do 循环以循环条件结尾，而不是标记循环主体结束的花括号。下面的 do 循环和前面的 while 循环打印相同的结果：

```
int count = 0;
do {
   System.out.println(count);
   count++;
} while(count < 10);
```

do 循环比类似的 while 循环少见得多，因为在实际使用中很少遇到一定会至少先执行一次循环的情况。

2.5.10 for语句

for 语句提供的循环结构往往比 while 和 do 循环更便利。for 语句利用了一般循环的执行模式。大多数循环都有一个计数器，或者某种形式的状态变量，在循环开始前初始化，然后测试这个变量的值，决定是否执行循环主体，再次计算表达式的值之前，在循环主体末尾递增或者以某种方式更新这个变量的值。初始化、测试和更新，这三步是循环变量的重要操作，for 语句把这三步作为循环句法的明确组成部分：

```
for(initialize; test; update) {
    statement
}
```

for 循环基本等同于下面的 while 循环：

```
initialize;
while (test) {
   statement;
   update;
}
```

把 initialize、test 和 update 三个表达式放在 for 循环的开头，特别有助于理解循环的作用，还能避免一些错误，例如忘记初始化或更新循环变量。解释器会丢掉 initialize 和 update 两个表达式的返回值，所以它们必须有副作用。initialize 一般是赋值表达式，update 一般是递增、递减或其他赋值表达式。

下面的 for 循环与前面的 while 和 do 循环一样，打印数字 0 到 9：

```
int count;
for(count = 0 ; count < 10 ; count++)
   System.out.println(count);
```

注意，这种句法把循环变量的重要信息都放在同一行，更能看清循环的执行方式。而且，把更新循环变量的表达式放在 for 语句中，还简化了循环主体，只剩一个语句，甚至不需要使用花括号组成语句块。

for 循环还支持一种句法，可以让循环更便于使用。很多循环都只在循环内部使用循环变量，因此 for 循环允许 initialize 是一个完整的变量声明表达式，这样循环变量的作用域

是循环主体，在循环外部不可见。例如：

```
for(int count = 0 ; count < 10 ; count++)
   System.out.println(count);
```

而且，for 循环的句法不限制只能使用一个变量，initialize 和 update 表达式都能使用逗号分隔多个初始化和更新表达式。例如：

```
for(int i = 0, j = 10 ; i < 10 ; i++, j--)
     sum += i * j;
```

在目前所举的例子中，计数器都是数字，但 for 循环并不限制计数器只能使用数字。例如，可以使用 for 循环迭代链表中的元素：

```
for(Node n = listHead; n != null; n = n.nextNode())
   process(n);
```

for 循环中的 initialize、test 和 update 表达式都是可选的，只有分隔这些表达式的分号是必须的。如果没有 test 表达式，其值假定为 true。因此，可以使用 for(;;) 编写一个无限循环。

2.5.11 遍历语句

Java 的 for 循环能很好地处理基本类型，但处理对象集合时没什么用，而且笨拙。不过，有种叫作“遍历循环”（foreach loop）的句法可以处理需要循环的对象集合。

遍历循环以关键字 for 开头，后面跟着一对括号，括号里是变量声明（不初始化）、冒号和表达式，括号后面是组成循环主体的语句（或语句块）：

```
for( declaration : expression )
     statement
```

别被遍历循环这个名字迷惑了，它并不使用关键字 foreach。冒号一般读作“……中”，例如“studentNames 中的各个名字”。

介绍 while、do 和 for 循环时，都举了一个例子，打印 10 个数字。遍历循环也能做到，但需要迭代一个集合。为了循环 10 次（打印 10 个数字），我们需要一个有 10 个元素的数组或其他集合。我们可以使用下面的代码：

```
// 这些是我们想打印的数字
int[] primes = new int[] { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };
// 这是打印这些数字的循环
for(int n : primes)
     System.out.println(n);
```

遍历不能做的事

遍历和 while、for 或 do 循环不同，因为它隐藏了循环计数器或 Iterator 对象。介绍 lambda 表达式时会看出来，这种想法很好，但有些算法不能使用遍历循环自然地表达出来。

假如你想把数组中的元素打印出来，各元素使用逗号分隔。为此，要在数组的每个元素后面打印一个逗号，但最后一个元素后面没有逗号；或者说，数组的每个元素前面都要打印一个逗号，但第一个元素前面没有逗号。使用传统的 for 循环，代码可以这样写：

```
for(int i = 0; i < words.length; i++) {
     if (i > 0) System.out.print(", ");
     System.out.print(words[i]);
}
```

这是很简单的任务，但遍历做不到，因为遍历循环没有循环计数器，也没有其他能识别第一次、最后一次或中间某次迭代的方式。

> 使用遍历循环迭代集合中的元素也有类似的问题。使用遍历循环迭代数组时无法获取当前元素的索引，同样，使用遍历循环迭代集合也无法获取列举集合元素的 Iterator 对象。

还有一些事情遍历循环做不到：

- 反向迭代数组或 List 对象中的元素；
- 使用同一个循环计数器获取两个不同数组同一索引位的元素；
- 调用 List 对象的 get() 方法无法迭代其中的元素，必须调用 List 对象的迭代器。

2.5.12 break语句

break 语句让 Java 解释器立即跳出所在的语句块。我们已经见过 break 语句在 switch 语句中的用法。break 语句最常写成关键字 break 后跟一个分号：

```
break;
```

这种形式让 Java 解释器立即退出所在的最内层 while、do、for 或 switch 语句。例如：

```
for(int i = 0; i < data.length; i++) {
    if (data[i] == target) {  // 找到需要的数据时
        index = i;              // 记住数据所在的位置
        break;                  // 然后停止查找
    }
}   // 执行break语句后,Java解释器来到这里
```

break 语句后面也可以跟着标注语句的名称。此时，break 语句让 Java 解释器立即退出指定的语句块。退出的语句块可以是任何类型，不只局限于循环或 switch 语句。例如：

```
TESTFORNULL: if (data != null) {
   for(int row = 0; row < numrows; row++) {
     for(int col = 0; col < numcols; col++) {
       if (data[row][col] == null)
         break TESTFORNULL;          // 把数组当成未定义
     }
   }
}  // 执行break TESTFORNULL语句后,Java解释器来到这里
```

2.5.13 continue语句

break 语句的作用是退出循环，而 continue 语句的作用是中止本次循环，开始下一次循环。continue 语句，不管是无标注还是有标注形式，只能在 while、do 或 for 循环中使用。如果没指定标注，continue 语句让最内层循环开始下一次循环；如果指定了标注，continue 语句让对应的循环开始下一次循环。例如：

```
for(int i = 0; i < data.length; i++) { // 循环处理数据
   if (data[i] == -1)                  // 如果缺失某个数据
     continue;                         // 跳到下一次循环
   process(data[i]);                   // 处理数据
}
```

在 while、do 和 for 循环中，continue 语句开始下一次循环的方式稍有不同。

- 在 while 循环中，Java 解释器直接返回循环开头，再次测试循环条件，如果计算结果为 true，再次执行循环主体。
- 在 do 循环中，解释器跳到循环的末尾，测试循环条件，决定是否要执行下一次循环。
- 在 for 循环中，解释器跳到循环的开头，先计算 update 表达式，然后计算 test 表达式，以此决定是否继续循环。由示例可以看出，有 continue 语句的 for 循环和基本等效的 while 循环行为不同：在 for 循环中会计算 update 表达式，而在等效的 while 循环中不会计算。

2.5.14 return语句

return 语句告诉 Java 解释器，终止执行当前方法。如果声明方法时指明了有返回值，return 语句后面必须跟着一个表达式。这个表达式的返回值就是这个方法的返回值。例如，下述方法计算并返回一个数字的平方：

```
double square(double x) {     // 计算x平方的方法
   return x * x;              // 计算并返回一个值
}
```

有些方法声明时使用了 void，指明不返回任何值。Java 解释器运行这种方法时，会依次执行其中的语句，直到方法结束为止。执行完最后一个语句时，解释器隐式返回。然而，有时没有返回值的方法要在到达最后一个语句之前显式返回。此时，可以使用后面没有任何

表达式的 return 语句。例如，下述方法只打印不返回参数的平方根。如果参数是负数，直接返回，不打印任何内容：

```
// 打印x平方根的方法
void printSquareRoot(double x) {
   if (x < 0) return;               // 如果x是负数,返回
   System.out.println(Math.sqrt(x)); // 打印x的平方根
}                                    // 方法结束,隐式返回
```

2.5.15　synchronized语句

Java 一直支持多线程编程，后文会详细介绍这个话题（尤其是 6.5 节）。不过读者要注意，并发编程不容易，有很多难以捉摸的地方。

具体而言，处理多线程时，经常必须避免多个线程同时修改同一个对象，以防对象的状态有冲突。Java 提供的 synchronized 语句可以帮助程序员，避免发生冲突。synchronized 语句的句法为：

```
synchronized ( expression ) {
   statements
}
```

expression 表达式的计算结果必须是一个对象或数组。statements 是能导致破坏的代码块，必须放在花括号里。

执行语句块之前，Java 解释器先为 expression 计算得到的对象或数组获取一个排它锁（exclusive lock），直到语句块执行完毕后再释放。只要某个线程拥有对象的排它锁，其他线程就不能再获取这个锁。

在 Java 中，synchronized 关键字还可以作为方法的修饰符。应用于方法时，synchronized 关键字指明整个方法都被锁定。如果 synchronized 关键字应用于类方法（静态方法），执行方法前，Java 会先为这个类获取一个排它锁。如果 synchronized 关键字应用于实例方法，Java 为类的实例获取一个排它锁。（类和实例在第 3 章介绍。）

2.5.16　throw语句

异常是一种信号，表明发生了某种异常状况或错误。抛出异常的目的是发出信号，表示有异常状况发生。捕获异常的目的是处理异常，使用必要的操作修复。在 Java 中，throw 语句用于抛出异常：

```
throw expression;
```

expression 的计算结果必须是一个异常对象，说明发生了什么异常或错误。稍后会详细介绍异常的种类，现在你只需知道，异常通过有点特殊的对象表示。下面是抛出异常的示例

代码：

```
public static double factorial(int x) {
   if (x < 0)
     throw new IllegalArgumentException("x must be >= 0");
   double fact;
   for(fact=1.0; x > 1; fact *= x, x--)
     /* empty */ ;          // 注意,使用的是空语句
   return fact;
}
```

Java 解释器执行 throw 语句时，会立即停止常规的程序执行，开始寻找能捕获或处理异常的异常处理程序。异常处理程序使用 try/catch/finally 语句编写，下一节会介绍。Java 解释器先在当前代码块中查找异常处理程序，如果有，解释器会退出这个代码块，开始执行异常处理代码。异常处理程序执行完毕后，解释器会继续执行处理程序后的语句。

如果当前代码块中没有适当的异常处理程序，解释器会在外层代码块中寻找，直到找到为止。如果方法中没有能处理 throw 语句抛出的异常的异常处理程序，解释器会停止运行当前方法，返回调用这个方法的地方，开始在调用方法的代码块中寻找异常处理程序。Java 通过这种方式，通过方法的词法结构不断向上冒泡，顺着解释器的调用堆栈一直向上寻找。如果一直没有捕获异常，就会冒泡到程序的 main() 方法。如果在 main() 方法中也没有处理异常，Java 解释器会打印一个错误消息，还会打印一个堆栈跟踪，指明这个异常在哪里发生，然后退出。

2.5.17 try/catch/finally语句

Java 有两种稍微不同的异常处理机制。经典形式是使用 try/catch/finally 语句。这个语句的 try 子句是可能抛出异常的代码块。try 代码块后面是零个或多个 catch 子句，每个子句用于处理特定类型的异常，而且能处理多个不同类型的异常。如果 catch 块要处理多个异常，使用 | 符号分隔各个不同的异常。catch 子句后面是一个可选的 finally 块，包含清理代码，不管 try 块中发生了什么，始终都会执行。

> **try 块的句法**
>
> catch 和 finally 子句都是可选的，但每个 try 块都必须有这两个子句中的一个。try、catch 和 finally 块都放在花括号里。花括号是句法必须的一部分，即使子句只包含一个语句也不能省略。

下述代码演示了 try/catch/finally 语句的句法和作用：

```
try {
   // 正常情况下,这里的代码从上到下运行,没有问题
   // 但是,有时可能抛出异常
   // 可能是throw语句直接抛出
```

```
    // 也可能是调用的方法间接抛出
}
catch (SomeException e1) {
    // 这段代码中的语句用于处理SomeException或其子类类型的异常对象
    // 在这段代码中,可以使用名称e1引用那个异常对象
}
catch (AnotherException | YetAnotherException e2) {
    // 这段代码中的语句用于处理AnotherException、YetAnotherException
    // 或二者的子类类型的异常。在这段代码中,使用名称e2引用传入的异常对象
}
finally {
    // 不管try子句的结束方式如何,这段代码中的语句都会执行:
    //    1)正常结束:到达块的末尾
    //    2)由break、continue或return语句导致
    //    3)抛出异常,由上述catch子句处理
    //    4)抛出异常,未被捕获处理
    // 但是,如果在try子句中调用了System.exit(),解释器会立即退出
    // 不执行finally子句
}
```

1. try子句

try 子句的作用很简单，组建一段代码，其中有异常需要处理，或者因某种原因终止执行后需要使用特殊的代码清理。try 子句本身没什么用，异常处理和清理操作在 catch 和 finally 子句中进行。

2. catch子句

try 块后面可以跟着零个或多个 catch 子句，指定处理各种异常的代码。每个 catch 子句只有一个参数（可以使用特殊的 | 句法指明 catch 块能处理多种异常类型），指定这个子句能处理的异常类型，以及一个名称，用来引用当前处理的异常对象。catch 块能处理的类型必须是 Throwable 的子类。

有异常抛出时，Java 解释器会寻找一个 catch 子句，它的参数要和异常对象的类型相同，或者是这个类型的子类。解释器会调用它找到的第一个这种 catch 子句。catch 块中的代码应该执行处理异常状况所需的任何操作。假如异常是 java.io.FileNotFoundException，此时或许要请求用户检查拼写，然后重试。

不是所有可能抛出的异常都要有一个 catch 子句处理，有些情况下，正确的处理方式是让异常向上冒泡，由调用方法捕获。还有些情况，例如表示程序错误的 NullPointerException 异常，正确的处理方式或许是完全不捕获，随它冒泡，让 Java 解释器退出，打印堆栈跟踪和错误消息。

3. finally子句

finnaly 子句放在 try 子句后面，一般用来执行清理操作（例如关闭文件和网络连接）。finally 子句很有用，因为不管 try 块中的代码以何种方式结束执行，只要有代码执行，finally 子句中的代码就会执行。事实上，只有一种方法能让 try 子句退出而不执行

finally 子句——调用 System.exit() 方法，让 Java 解释器停止运行。

正常情况下，执行到 try 块的末尾后会继续执行 finally 块，做必要的清理工作。如果因为 return、continue 或 break 语句而离开 try 块，会先执行 finally 块，然后再转向新的目标代码。

如果 try 块抛出了异常，而且有处理该异常的 catch 块，那么先执行 catch 块，然后在执行 finally 块。如果本地没有能处理该异常的 catch 块，先执行 finally 块，然后再向上冒泡到能处理该异常最近的 catch 子句。

如果 finally 块使用 return、continue、break 或 throw 语句，或者调用的方法抛出了异常，从而转移了控制权，那么待转移的控制权中止，改为执行新的控制权转移。例如，如果 finally 子句抛出了异常，这个异常会取代任何正在抛出的异常。如果 finally 子句使用了 return 语句，就算抛出的异常还没处理，方法也会正常返回。

try 和 finally 子句可以放在一起使用，不处理异常，也没有 catch 子句。此时，finally 块只是负责清理的代码，不管 try 子句中有没有 break、continue 或 return 语句，都会执行。

2.5.18　处理资源的try语句

try 块的标准形式很通用，但有些常见的情况需要开发者小心编写 catch 和 finally 块。这些情况是清理或关闭不再需要使用的资源。

Java（从第 7 版起）提供了一种很有用的机制，能自动关闭需要清理的资源——处理资源的 try 语句（try-with-resources，TWR）。10.1 节会详细介绍 TWR，但为了本节的完整，先介绍它的句法。下面的示例展示了如何使用 FileInputStream 类打开文件（得到的对象需要清理）：

```
try (InputStream is = new FileInputStream("/Users/ben/details.txt")) {
  // ……处理这个文件
}
```

这种新型 try 语句的参数都是需要清理的对象。[2] 这些对象的作用域在 try 块中，不管 try 块以何种方式退出，都会自动清理。开发者无需编写任何 catch 或 finally 块，Java 编译器会自动插入正确的清理代码。

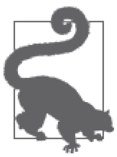
所有处理资源的新代码都应该使用 TWR 形式编写，因为这种形式比自己动手编写 catch 块更少出错，而且不会遇到麻烦的技术问题，例如终结（详情参见 6.4 节）。

注 2：严格来说，这些对象必须实现 AutoCloseable 接口。

2.5.19　assert语句

`assert` 语句用来验证 Java 代码的设计假想。断言（assertion）由 `assert` 关键字和布尔表达式组成，程序员认为布尔表达式的计算结果始终应该为 `true`。默认情况下断言未启用，`assert` 语句什么作用也没有。

不过，可以启用断言，作为一种调试工具。启用后，`assert` 语句会计算表达式。如果表达式的计算结果确是 `true`，`assert` 语句什么也不做；如果计算结果是 `false`，断言失败，`assert` 语句抛出 `java.lang.AssertionError` 异常。



在 JDK 库之外，极少使用 `assert` 语句。用它测试大多数应用都不灵便，一般的开发者很少使用，不过有时用来现场调试复杂的多线程应用。

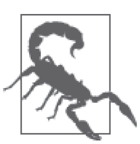
`assert` 语句可以包含可选的第二个表达式，使用冒号和第一个表达式分开。如果启用了断言，而且第一个表达式的计算结果为 `false`，那么第二个表达式的值会作为错误代码或错误消息传给 `AssertionError()` 构造方法。`assert` 语句的完整句法如下：

```
assert assertion;
```

或者：

```
assert assertion : errorcode;
```

为了有效使用断言，必须注意几处细节。首先，要记住，一般情况下程序没有启用断言，只有少数情况才会启用。这意味着，编写断言表达式时要小心，不能有副作用。



绝不要在自己编写的代码中抛出 `AssertionError` 异常，如果这么做，可能会在 Java 平台未来的版本中得到意料之外的结果。

如果抛出了 `AssertionError` 异常，表明程序员的假想之一没有实现。这意味着，在设计的使用范围之外使用了代码，无法正常运行。简单来说，没有看似合理的方式能从 `AssertionError` 异常中恢复，因此不要尝试捕获这个异常（除非在顶层简单捕获，以对用户更友好的方式显示错误）。

启用断言

为了效率，不应该在每次执行代码时都测试断言，因为 `assert` 语句认为假想始终为真。因此，默认情况下禁用了断言，`assert` 语句没有作用。不过，断言代码还是会编译到类文件

中，所以需要诊断或调试时可以启用断言。断言可以全局启用，也可以把命令行参数传给 Java 解释器，有选择性地启用。

如果想为系统类之外的所有类启用断言，使用 `-ea` 参数。如果想为系统类启用断言，使用 `-esa` 参数。如果想为某个具体的类启用断言，使用 `-ea` 参数，后跟一个冒号和类名：

```
java -ea:com.example.sorters.MergeSort com.example.sorters.Test
```

如果想为包中所有的类和子包启用断言，在 `-ea` 参数后面加上冒号、包名和三个点号：

```
java -ea:com.example.sorters... com.example.sorters.Test
```

使用 `-da` 参数，通过相同的方式可以禁用断言。例如，为整个包启用断言，但在某个类或子包中禁用，可以这么做：

```
java -ea:com.example.sorters... -da:com.example.sorters.QuickSort
java -ea:com.example.sorters... -da:com.example.sorters.plugins..
```

最后，类加载时可以控制是否启用断言。如果在程序中使用自定义的类加载程序（第 11 章会详细介绍自定义类加载），而且想启用断言，可能会对这些方法感兴趣。

2.6 方法

*方法*是有名称的 Java 语句序列，可被其他 Java 代码调用。调用方法时，可以传入零个或多个值，这些值叫*参数*。方法执行一些计算，还可以返回一个值。2.4 节介绍过，方法调用是 Java 解释器计算的表达式。不过，因为方法调用可以有副作用，因此，也能作为表达式语句使用。本节不讨论方法调用，只说明如何定义方法。

2.6.1 定义方法

你已经知道如何定义方法的主体了，方法主体就是放在花括号里的任意语句序列。更有趣的是方法的*签名*（signature）。[3] 签名指定下述内容：

- 方法的名称；
- 方法所用参数的数量、顺序、类型和名称；
- 方法的返回值类型；
- 方法能抛出的已检异常（签名还能列出未检异常，不过不是必需的）；
- 提供方法额外信息的多个方法修饰符。

方法签名定义了调用方法之前需要知道的一切信息，是方法的*规范*，而且定义了方法的

注 3：在 Java 语言规范中，术语“signature”有技术层面的意义，和这里使用的稍有不同。本书使用方法签名较不正式的定义。

API。若想使用 Java 平台的在线 API 参考指南，需要知道如何阅读方法签名。若想编写 Java 程序，需要知道如何定义自己的方法。方法都以方法签名开头。

方法签名的格式如下：

```
modifiers type name ( paramlist ) [ throws exceptions ]
```

签名（方法规范）后面是方法主体（方法的实现），即放在花括号里的 Java 语句序列。*抽象方法*（参见第 3 章）没有实现部分，方法主体使用一个分号表示。

方法签名中可能包含类型变量声明，这种方法叫*泛型方法*（generic method）。泛型方法和类型变量在第 4 章介绍。

下面是一些方法定义示例，都以签名开头，后面跟着方法主体：

```
// 这个方法传入的是字符串数组,没有返回值
// 所有Java程序的入口都是这个名称和签名
public static void main(String[] args) {
     if (args.length > 0) System.out.println("Hello " + args[0]);
     else System.out.println("Hello world");
}

// 这个方法传入两个double类型的参数,返回一个double类型的数字
static double distanceFromOrigin(double x, double y) {
     return Math.sqrt(x*x + y*y);
}

// 这是抽象方法,没有主体
// 注意,调用这个方法时可能会抛出异常
protected abstract String readText(File f, String encoding)
    throws FileNotFoundException, UnsupportedEncodingException;
```

`modifiers` 是零个或多个特殊的修饰符关键字，之间使用空格分开。例如，声明方法时可以使用 `public` 和 `static` 修饰符。允许使用的修饰符及其意义在下一节介绍。

方法签名中的 `type` 指明方法返回值的类型。如果方法没有返回值，`type` 必须是 `void`。如果声明方法时指定了返回类型，就必须包含一个 `return` 语句，返回一个符合（或能转换为）所声明类型的值。

*构造方法*是一段类似方法的代码，用于初始化新建的对象。第 3 章会介绍，构造方法的定义方式和方法类似，不过签名中没有 `type` 部分。

方法的修饰符和返回值类型后面是 `name`，即方法名。方法名和变量名一样，也是 Java 标识符。和所有 Java 标识符一样，方法名可以包含 Unicode 字符集能表示的任何语言的字母。定义多个同名方法是合法的，往往也很有用，只要各方法的参数列表不同就行。定义多个同名方法叫*方法重载*（method overloading）。

和某些其他语言不同，Java 没有匿名方法。不过，Java 8 引入了 lambda 表达式，作用类似于匿名方法，但是 Java 运行时会自动把 lambda 表达式转换成适当的具名方法，详情参见 2.7.5 节。

例如，我们见过的 `System.out.println()` 方法就是重载方法。具有这个名字的某个方法打印字符串，而具有这个名字的其他方法打印各种基本类型的值。Java 编译器根据传入这个方法的参数类型决定调用哪个方法。

定义方法时，方法名后一定是方法的形参列表（parameters list），而且必须放在括号里。形参列表定义零个或多个传入方法的实参（argument）。[4] 如果有形参的话，每个形参都包含类型和名称，（如果有多个形参）形参之间使用逗号分开。调用方法时，传入的实参值必须和该方法签名中定义的形参数量、类型和顺序匹配。传入的值不一定要和签名中指定的类型一样，但是必须能不经校正转换为对应的类型。

如果 Java 方法没有实参，其形参列表是 `()`，而不是 `(void)`。C 和 C++ 程序员要特别注意，Java 不把 `void` 当作一种类型。

Java 允许程序员定义和调用参数数量不定的方法，使用的句法叫*变长参数*（varargs），本章后面会详细介绍。

方法签名的最后一部分是 `throws` 子句，列出方法能抛出的*已检异常*（checked exception）。已检异常是一系列异常类，必须在能抛出它们的方法中使用 `throws` 子句列出。如果方法使用 `throw` 语句抛出一个已检异常，或者调用的其他方法抛出一个没有捕获或处理的已检异常，声明这个方法时就必须指明能抛出这个异常。如果方法能抛出一个或多个已检异常，要在参数列表后面使用 `throws` 关键字指明能抛出的异常类。如果方法不会抛出异常，无需使用 `throws` 关键字。如果方法抛出的异常类型不止一个，要使用逗号分隔异常类的名称。稍后还会再说明。

2.6.2 方法修饰符

方法的修饰符包含零个或多个修饰符关键字，例如 `public`、`static` 或 `abstract`。下面列出允许使用的修饰符及其意义。

注 4：parameter 是定义方法时声明的参数，argument 是调用方法时传入的参数。如果二者同时出现，parameter 译为“形参”，argument 译为“实参”。在不引起歧义的情况下，则都译为“参数”。
——译者注

- abstract

  使用 abstract 修饰的方法没有实现主体。组成普通方法主体的花括号和 Java 语句使用一个分号代替。如果类中有使用 abstract 修饰的方法，类本身也必须使用 abstract 声明。这种类不完整，不能实例化（参见第 3 章）。

- final

  使用 final 修饰的方法不能被子类覆盖或隐藏，能获得普通方法无法得到的编译器优化。所有使用 private 修饰的方法都隐式添加了 final 修饰符；使用 final 声明的任何类，其中的所有方法也都隐式添加 final 修饰符。

- native

  native 修饰符表明方法的实现使用某种“本地”语言编写，例如 C 语言，并且开放给 Java 程序使用。native 修饰的方法和 abstract 修饰的方法一样，没有主体：花括号使用一个分号代替。

实现 native 修饰的方法

Java 刚出现时，使用 native 修饰方法有时是为了提高效率。现在几乎不需要这么做了。现在，使用 native 修饰方法的目的是，把 Java 代码集成到现有的 C 或 C++ 库中。native 修饰的方法和所在平台无关，如何把实现和方法声明所在的 Java 类链接起来，取决于 Java 虚拟机的实现方式。本书没有涵盖使用 native 修饰的方法。

- public、protected、private

  这些访问修饰符指定方法是否能在定义它的类之外使用，或者能在何处使用。这些非常重要的修饰符在第 3 章说明。

- static

  使用 static 声明的方法是*类方法*，关联在类自己身上，而不是类的实例身上（第 3 章会详细说明）。

- strictfp

  在这个很少使用的奇怪修饰符中，fp 的意思是“浮点”（floating point）。一般情况下，Java 会利用运行时所在平台的浮点硬件提供的可用扩展精度。添加这个关键字后，运行 strictfp 修饰的方法时，Java 会严格遵守标准，而且就算结果不精确，也只使用 32 位或 64 位浮点数格式进行浮点运算。

- synchronized

  synchronized 修饰符的作用是实现线程安全的方法。线程调用 synchronized 修饰的方法之前，必须先为方法所在的类（针对 static 修饰的方法）或对应的类实例（针对没使用 static 修饰的方法）获取一个锁，避免两个线程同时执行该方法。

synchronized 修饰符是实现的细节（因为方法可以通过其他方式实现线程安全），不是方法规范或 API 的正式组成部分。好的文档应该明确说明方法是否线程安全，使用多线程程序时不能依赖于是否有 synchronized 关键字。

注解是特例（注解的详细介绍参见第 4 章）——注解可以看作方法修饰符和额外补充信息的折中方案。

2.6.3　已检异常和未检异常

Java 的异常处理机制会区分两种不同的异常类型：*已检异常*和*未检异常*。

已检异常和未检异常之间的区别在于异常在什么情况下抛出。已检异常在明确的特定情况下抛出，经常是应用能部分或完全恢复的情况。

例如，某段代码要在多个可能的目录中寻找配置文件。如果试图打开的文件不在某个目录中，就会抛出 FileNotFoundException 异常。在这个例子中，我们想捕获这个异常，然后在文件可能出现的下一个位置继续尝试。也就是说，虽然文件不存在是异常状况，但可以从中恢复，这是意料之中的失败。

然而，在 Java 环境中有些失败是无法预料的，这些失败可能是由运行时条件或滥用库代码导致的。例如，无法正确预知 OutOfMemoryError 异常；又如，把无效的 null 传给使用对象或数组的方法，会抛出 NullPointerException 异常。

这些是未检异常。基本上任何方法在任何时候都可能抛出未检异常。这是 Java 环境中的墨菲定律："会出错的事总会出错。"从未检异常中恢复，虽说不是不可能，但往往很难，因为完全不可预知。

若想区分已检异常和未检异常，记住两点：异常是 Throwable 对象，而且异常主要分为两类，通过 Error 和 Exception 子类标识。只要异常对象是 Error 类，就是未检异常。Exception 类还有一个子类 RuntimeException，RuntimeException 类的所有子类都属于未检异常。除此之外，都是已检异常。

处理已检异常

Java 为已检异常和未检异常制定了不同的规则。如果定义的方法会抛出已检异常，就必须在方法签名的 throws 子句中声明这个异常。Java 编译器会检查方法签名，确保的确声明了；如果没声明，会导致编译出错（所以才叫"已检异常"）。

就算自己从不抛出已检异常，有时也必须使用 throws 子句声明已检异常。如果方法中调用

了会抛出已检异常的方法，要么加入异常处理代码处理这个异常，要么使用 throws 子句声明这个方法也能抛出这个异常。

例如，下述方法使用标准库中的 java.net 和 URL 类（第 10 章会介绍）访问网页，尝试估算网页的大小。所用的方法和构造方法会抛出各种 java.io.IOException 异常对象，所以在 throws 子句中声明了：

```
public static estimateHomepageSize(String host) throws IOException {
    URL url = new URL("htp://"+ host +"/");
    try (InputStream in = url.openStream()) {
        return in.available();
    }
}
```

其实，上述代码有个问题：协议名拼写错了——没有名为 htp:// 的协议。所以，estimateHomepageSize() 方法会一直失败，抛出 MalformedURLException 异常。

你怎么知道要调用的方法会抛出已检异常呢？可以查看这个方法的签名。如果签名中没有，但又必须处理或声明调用的方法抛出的异常时，Java 编译器会（通过编译错误消息）告诉你。

2.6.4　变长参数列表

方法可以声明为接受数量不定的参数，调用时也可以传入数量不定的参数。这种方法一般叫作*变长参数方法*。格式化打印方法 System.out.printf() 和 String 类相关的 format() 方法，以及 java.lang.reflect 中反射 API 的一些重要方法，都使用变长参数。

变长参数列表的声明方式为，在方法最后一个参数的类型后面加上省略号（...），指明最后一个参数可以重复零次或多次。例如：

```
public static int max(int first, int... rest) {
    /* 暂时省略主体 */
}
```

变长参数方法纯粹由编译器处理，把数量不定的参数转换为一个数组。对 Java 运行时来说，上面的 max() 方法和下面这个没有区别：

```
public static int max(int first, int[] rest) {
    /* 暂时省略主体 */
}
```

把变长参数方法的签名转换为真正的签名，只需把 ... 换成 []。记住，参数列表中只能有一个省略号，而且只能出现在最后一个参数中。

下面填充 max() 方法的主体：

```
public static int max(int first, int... rest) {
    int max = first;
    for(int i : rest) { // 合法,因为rest其实就是数组
        if (i > max) max = i;
    }
    return max;
}
```

声明这个 max() 方法时指定了两个参数，第一个是普通的 int 类型值，但是第二个可以重复零次或多次。下面对 max() 方法的调用都是合法的：

```
max(0)
max(1, 2)
max(16, 8, 4, 2, 1)
```

因为变长参数方法被编译成接受数组参数的方法，所以在编译对这类方法的调用得到的代码中，包含创建和初始化这个数组的代码。因此，调用 max(1,2,3) 被编译成：

```
max(1, new int[] { 2, 3 })
```

其实，如果参数的方法已经存储在数组中，完全可以直接把数组传给变长参数方法，而不用把数组中的元素取出来一个一个传入。... 参数可以看成一个数组。不过，反过来就不行了：只有使用省略号声明为变长参数方法，才能使用变长参数方法调用的句法。

2.7 介绍类和对象

我们已经介绍了运算符、表达式、语句和方法，终于可以介绍类了。*类*是一段代码的名称，其中包含很多保存数据值的字段和操作这些值的方法。类是 Java 支持的五种引用类型之一，而且是最重要的一种。我们会在单独的一章（第 3 章）全面介绍类。这里之所以要介绍，是因为类是继方法之后的另一种高级句法，而且本章剩下的内容需要对类的概念有基本的认识，要知道定义类、实例化类和使用所得*对象*的基本句法。

关于类最重要的事情是，它们定义了一种新数据类型。例如，可以定义一个名为 Point 的类，表示笛卡尔二维坐标系中的数据点。这个类可能会定义两个字段，保存点的 *x* 和 *y* 坐标，还可能会定义处理和操作点的方法。Point 类就是一个新数据类型。

谈论数据类型时，要把数据类型和数据类型表示的值区分开，这一点很重要。char 是一种数据类型，用于表示 Unicode 字符。但是一个 char 类型的值表示某个具体的字符。类是一种数据类型，而类表示的值是*对象*。我们使用“类”这个名称的原因是，每个类定义一种对象。Point 类是一种数据类型，用于表示 (*x*, *y*) 点，而 Point 对象表示某个具体的 (*x*, *y*) 点。正如你想得那样，类和类的对象联系紧密。在接下来的几节中，会介绍这两个概念。

2.7.1 定义类

前面讨论的 `Point` 类可以使用下面的方式定义：

```
/** 表示笛卡尔坐标系中的(x,y)点 */
public class Point {
    // 点的坐标
    public double x, y;
    public Point(double x, double y) {    // 初始化字段的构造方法
        this.x = x; this.y = y;
    }

    public double distanceFromOrigin() {  // 操作x和y字段的方法
        return Math.sqrt(x*x + y*y);
    }
}
```

这个类的定义保存在一个名为 Point.java 的文件中，然后编译成一个名为 Point.class 的文件，供 Java 程序和其他类使用。现在定义这个类只是为了完整性，并提供上下文，不要奢望能完全理解所有细节。第 3 章的大部分内容会专门讲解如何定义类。

记住，你不需要定义想在 Java 程序中使用的每个类。Java 平台包含上千个预先定义好的类，在每台运行 Java 的电脑中都能使用。

2.7.2 创建对象

我们已经定义了 `Point` 类，现在 `Point` 是一种新数据类型，我们可以使用下面的代码声明一个变量，存储一个 `Point` 对象：

```
Point p;
```

不过，声明一个存储 `Point` 对象的变量并不会创建这个对象。要想创建对象，必须使用 `new` 运算符。这个关键字后面跟着对象所属的类（即对象的类型）和括号中可选的参数列表。这些参数会传入类的构造方法，初始化新对象的内部字段：

```
// 创建一个Point对象,表示(2,-3.5)
// 声明一个变量p,存储这个新Point对象的引用
Point p = new Point(2.0, -3.5);

// 创建一些其他类型的对象
// 一个Date对象,表示当前时间
Date d = new Date();
// 一个HashSet对象,保存一些对象
Set words = new HashSet();
```

`new` 关键字是目前为止在 Java 中创建对象最常用的方式。还有一些其他方式也有必要提一下。首先，有些符合特定条件的类很重要，Java 为这些类定义了专用的字面量句法，用于创建这些类型的对象（本节后面会介绍）。其次，Java 支持动态加载机制，允许程序动态

加载类和创建类的实例，详情参见第 11 章。最后，对象还可以通过反序列化创建。对象的状态可以保存或序列化到一个文件中，然后可以使用 java.io.ObjectInputStream 类重新创建这个对象。

2.7.3　使用对象

我们已经知道如何定义类，如何通过创建对象实例化类，现在要介绍使用对象的 Java 句法。前面说过，类定义了一些字段和方法。每个对象都有自己的字段副本，而且可以访问类中的方法。我们使用点号（.）访问对象的具名字段和方法。例如：

```
Point p = new Point(2, 3);         // 创建一个对象
double x = p.x;                    // 读取这个对象的一个字段
p.y = p.x * p.x;                   // 设定一个字段的值
double d = p.distanceFromOrigin(); // 访问这个对象的一个方法
```

这种句法在面向对象语言中很常见，Java 也不例外，因此会经常见到。特别注意一下 p.distanceFromOrigin()。这个表达式告诉 Java 编译器，查找一个名为 distanceFromOrigin() 的方法（在 Point 类中定义），然后使用这个方法对 p 对象的字段进行计算。第 3 章会详细介绍这种操作。

2.7.4　对象字面量

介绍基本类型时我们看到，每种基本类型都有字面量句法，可以直接在程序的代码中插入各种类型的值。Java 还为一些特殊的引用类型定义了字面量句法，介绍如下。

1. 字符串字面量

String 类使用一串字符表示文本。因为程序经常需要通过文字和用户沟通，所以在任何编程语言中处理文本字符串的能力都十分重要。在 Java 中，字符串是对象，表示文本的数据类型是 String 类。现代 Java 程序使用的字符串数据通常比其他程序都多。

因为字符串是如此基本的数据类型，所以 Java 允许在程序中插入文本字面量，方法是把字符放在双引号（"）中。例如：

```
String name = "David";
System.out.println("Hello, " + name);
```

别把字符串字面量两侧的双引号和字符字面量两侧的单引号搞混了。字符串字面量可以包含字符字面量中能使用的任何一个转义序列（参见表 2-2）。在双引号包围的字符串字面量中嵌入双引号时，转义序列特别有用。例如：

```
String story = "\t\"How can you stand it?\" he asked sarcastically.\n";
```

字符串字面量中不能包含注释，而且只能有一行。Java 不支持把两行当成一行的任何接续

字符。如果需要表示一串长文本，一行写不下，可以把这个文本拆成多个单独的字符串字面量，再使用 + 运算符把它们连接起来。例如：

```
// 这么写不合法,字符串字面量不能断行
String x = "This is a test of the
            emergency broadcast system";

String s = "This is a test of the " +      // 要这么写
           "emergency broadcast system";
```

这种字面量连接在编译程序时，而不是运行时完成，所以无需担心性能会降低。

2. 类型字面量

第二种支持专用对象字面量句法的类型是 Class 类。Class 类的实例表示一种 Java 数据类型，而且包含所表示类型的元数据。若想在 Java 程序中使用 Class 对象字面量，要在数据类型的名称后面加上 .class。例如：

```
Class<?> typeInt = int.class;
Class<?> typeIntArray = int[].class;
Class<?> typePoint = Point.class;
```

3. null引用

null 关键字是一种特殊的字面量，引用不存在的值，或者不引用任何值。null 这个值是独一无二的，因为它是任何一种引用类型的成员。null 可以赋值给属于任何引用类型的变量。例如：

```
String s = null;
Point p = null;
```

2.7.5　lambda表达式

Java 8 引入了一个重要的新功能——lambda *表达式*。这是十分常见的编程语言结构，在*函数式编程语言*（Functional Programming Language，例如 Lisp、Haskell 和 OCaml）中使用范围极广。lambda 表达式的功能和灵活性远非局限于函数式语言，在几乎所有的现代编程语言中都能看到它的身影。

> **定义 lambda 表达式**
>
> lambda 表达式其实就是没有名称的函数，在 Java 中可以把它当成一个值。Java 不允许脱离类的概念运行方法，所以 lambda 表达式是在某个类中定义的匿名方法（开发者可能不知道具体是哪个类）。

lambda 表达式的句法如下：

```
( paramlist ) -> { statements }
```

下面是一个十分传统的简单示例：

```
Runnable r = () -> System.out.println("Hello World");
```

lambda 表达式当成值使用时，会根据要存储的变量类型，自动转换为相应的对象。自动转换和*类型推导*是 Java 实现 lambda 表达式的基础。但是，这要求正确地理解 Java 的整个类型系统。4.5 节会详细说明 lambda 表达式，现在只需知道句法。

下面是个稍微复杂的示例：

```
ActionListener listener = (e) -> {
  System.out.println("Event fired at: "+ e.getWhen());
  System.out.println("Event command: "+ e.getActionCommand());
};
```

2.8　数组

*数组*是一种特殊的对象，保存零个或多个基本类型或引用类型的值。这些值是数组的*元素*，是通过所在位置或*索引*引用的无名变量。数组的类型通过*元素的类型*表示，数组中的所有元素必须都属于这个类型。

数组元素的编号从零开始，有效的索引范围是零到元素数量减一。例如，索引为 1 的元素，是数组中的*第二个*元素。数组中的元素数量是数组的*长度*。数组的长度在创建时指定，从此就不能改变。

数组中元素的类型可以是任何有效的 Java 类型，包括数组类型。也就是说，Java 支持由数组组成的数组，实现多维数组。Java 不支持其他语言中的矩阵式多维数组。

2.8.1　数组的类型

数组的类型和类一样，也是引用类型。数组的实例和类的实例一样，也是对象。[5] 和类不同的是，数组的类型不用定义，只需在元素类型后面加上一对中括号即可。例如，下述代码声明了三种不同类型的数组：

```
byte b;                        // byte是基本类型
byte[] arrayOfBytes;           // byte[]是由byte类型的值组成的数组
byte[][] arrayOfArrayOfBytes;  // byte[][]是由byte[]类型的值组成的数组
String[] points;               // String[]是由字符串组成的数组
```

数组的长度不是数组类型的一部分。例如，声明一个方法，并且期望传入恰好由四个 `int` 类型的值组成的数组，是不可能的。如果方法的参数类型是 `int[]`，调用时传入的数组可

注 5：讨论数组时，有个术语上的难题。与类和类的实例不同，数组的类型和数组实例都使用“数组”这个术语表示。在实际使用中，一般通过上下文能分清讨论的是类型还是值。

以包含任意个元素（包括零个）。

数组类型不是类，但数组实例是对象。这意味着，数组从 java.lang.Object 类继承了方法。数组实现了 Cloneable 接口，而且覆盖了 clone() 方法，确保数组始终能被复制，而且 clone() 方法从不抛出 CloneNotSupportedException 异常。数组还实现了 Serializable 接口，所以只要数组中元素的类型能被序列化，数组就能被序列化。而且，所有数组都有一个名为 length 的字段，这个字段的修饰符是 public final int，表示数组中元素的数量。

1. 数组类型放大转换

因为数组扩展自 Object 类，而且实现了 Cloneable 和 Serializable 接口，所以任何数组类型都能放大转换成这三种类型中的任何一种。而且，特定的数组类型还能放大转换成其他数组类型。如果数组中的元素类型是引用类型 T，而且 T 能指定给类型 S，那么数组类型 T[] 就能指定给数组类型 S[]。注意，基本类型的数组不能放大转换。例如，下述代码展示了合法的数组放大转换：

```
String[] arrayOfStrings;      // 创建字符串数组
int[][] arrayOfArraysOfInt;   // 创建int二维数组
// String可以指定给Object,
// 因此String[]可以指定给Object[]
Object[] oa = arrayOfStrings;
// String实现了Comparable接口
// 因此String[]可以视作Comparable[]
Comparable[] ca = arrayOfStrings;
// int[]是Object类的对象,因此int[][]可以指定给Object[]
Object[] oa2 = arrayOfArraysOfInt;
// 所有数组都是可以复制和序列化的对象
Object o = arrayOfStrings;
Cloneable c = arrayOfArraysOfInt;
Serializable s = arrayOfArraysOfInt[0];
```

因为数组类型可以放大转换成另一种数组类型，所以编译时和运行时数组的类型并不总是一样。

> 这种放大转换叫作“数组协变”（array covariance）。从 4.2.5 节或许可以看出，现代标准认为这是历史遗留的不合理功能，因为编译时和运行时得出的类型不一致。

把引用类型的值存储在数组元素中之前，编译器通常必须插入运行时检查，确保运行时这个值的类型和数组元素的类型匹配。如果运行时检查失败，会抛出 ArrayStoreException 异常。

2. 与C语言兼容的句法

如前所示，指定数组类型的方法是在元素类型后加上一对中括号。为了兼容 C 和 C++，

Java 还支持一种声明变量的句法：中括号放在变量名后面，元素类型后面可以放也可以不放中括号。这种句法可用于局部变量，字段和方法的参数。例如：

```
// 这行代码声明类型为int,int[]和int[][]的局部变量
int justOne, arrayOfThem[], arrayOfArrays[][];

// 这三行代码声明的字段属于同一种数组类型
public String[][] aas1;   // 推荐使用的Java句法
public String aas2[][];   // C语言的句法
public String[] aas3[];   // 令人困惑的混用句法

// 这个方法签名包含两个类型相同的参数
public static double dotProduct(double[] x, double y[]) { ... }
```

这种兼容句法极其少见，不要使用。

2.8.2 创建和初始化数组

在 Java 中，使用 new 关键字创建数组，就像创建对象一样。数组类型没有构造方法，但创建数组时要指定长度，在中括号里使用非负整数指定所需的数组大小：

```
// 创建一个能保存1024个byte类型数据的新数组
byte[] buffer = new byte[1024];
// 创建一个能保存50个字符串引用的数组
String[] lines = new String[50];
```

使用这种句法创建的数组，每个元素都会自动初始化，初始值和类中的字段默认值相同：boolean 类型元素的初始值是 false，char 类型元素的初始值是 \u0000，整数元素的初始值是 0，浮点数元素的初始值是 0.0，引用类型元素的初始值是 null。

创建数组的表达式也能用来创建和初始化多维数组。这种句法稍微复杂一些，本节后面会介绍。

数组初始化程序

若想在一个表达式中创建数组并初始化其中的元素，不要指定数组的长度，在方括号后面跟着一对花括号，在花括号里写入一些逗号分隔的表达式。当然了，每个表达式的返回值类型必须能指定给数组元素的类型。创建的数组长度和表达式的数量相等。这组表达式的最后一个后面可以加上逗号，但没必要这么做。例如：

```
String[] greetings = new String[] { "Hello", "Hi", "Howdy" };
int[] smallPrimes = new int[] { 2, 3, 5, 7, 11, 13, 17, 19, };
```

注意，这种句法无需把数组赋值给变量就能创建、初始化和使用数组。某种意义上，这种创建数组的表达式相当于匿名数组字面量。下面是几个示例：

```
// 调用一个方法,传入一个包含两个字符串的匿名数组字面量
String response = askQuestion("Do you want to quit?",
                              new String[] {"Yes", "No"});

// 调用另一个方法,传入匿名对象组成的匿名数组
double d = computeAreaOfTriangle(new Point[] { new Point(1,2),
                                               new Point(3,4),
                                               new Point(3,2) });
```

如果数组初始化程序是变量声明的一部分，可以省略 new 关键字和元素类型，在花括号里列出所需的元素：

```
String[] greetings = { "Hello", "Hi", "Howdy" };
int[] powersOfTwo = {1, 2, 4, 8, 16, 32, 64, 128};
```

数组字面量在程序运行时，而不是程序编译时，创建和初始化。例如下述数组字面量：

```
int[] perfectNumbers = {6, 28};
```

编译得到的 Java 字节码和下面的代码相同：

```
int[] perfectNumbers = new int[2];
perfectNumbers[0] = 6;
perfectNumbers[1] = 28;
```

Java 在运行时初始化数组有个重要的推论：数组初始化程序中的表达式可能会在运行时计算，而且不一定非要使用编译时常量。例如：

```
Point[] points = { circle1.getCenterPoint(), circle2.getCenterPoint() };
```

2.8.3　使用数组

创建数组后就可以开始使用了。随后的几节说明访问元素的基本方法，以及常见的数组用法，例如迭代数组中的元素，复制数组或数组的一部分。

1. 访问数组中的元素

数组中的元素是变量。如果元素出现在表达式中，其计算结果是这个元素中保存的值。如果元素出现在赋值运算符的左边，会把一个新值保存到这个元素中。不过，元素和普通的变量不同，它没有名字，只有编号。数组中的元素使用方括号访问。假如 a 是一个表达式，其计算结果为一个数组引用，那么可以使用 a[i] 索引数组，并引用某个元素。其中，i 是整数字面量或计算结果为 int 类型值的表达式。例如：

```
// 创建一个由两个字符串组成的数组
String[] responses = new String[2];
```

```
responses[0] = "Yes";  // 设定数组的第一个元素
responses[1] = "No";   // 设定数组的第二个元素

// 读取这个数组中的元素
System.out.println(question + " (" + responses[0] + "/" +
                   responses[1] + " ): ");

// 数组引用和数组索引都可以是复杂的表达式
double datum = data.getMatrix()[data.row() * data.numColumns() +
                   data.column()];
```

数组的索引表达式必须是 `int` 类型，或能放大转换成 `int` 的类型：`byte`、`short`，甚至是 `char`。数组的索引显然不能是 `boolean`、`float` 或 `double` 类型。还记得吗，数组的 `length` 字段是 `int` 类型，所以数组中的元素数量不能超过 `Integer.MAX_VALUE`。如果使用 `long` 类型的表达式索引数组，即便运行时表达式的返回值在 `int` 类型的取值范围内，也会导致编译出错。

2. 数组的边界

还记得吗？数组 `a` 的第一个元素是 `a[0]`，第二个元素是 `a[1]`，最后一个元素是 `a[a.length-1]`。

使用数组时常见的错误是索引太小（负数）或太大（大于或等于数组的长度）。在 C 或 C++ 等语言中，如果访问起始索引之前或结尾索引之后的元素，会导致无法预料的行为，而且在不同的调用和不同的平台中有所不同。这种问题不一定会被捕获，如果没捕获，可能过一段时间才会发现。因为在 Java 中容易编写错误的索引代码，所以运行时每次访问数组都会做检查，确保得到能预料的结果。如果数组的索引太小或太大，Java 会立即抛出 `ArrayIndexOutOfBoundsException` 异常。

3. 迭代数组

为了在数组上执行某种操作，经常要编写循环，迭代数组中的每个元素。这种操作通常使用 `for` 循环完成。例如，下述代码计算整数数组中的元素之和：

```
int[] primes = { 2, 3, 5, 7, 11, 13, 17, 19, 23 };
int sumOfPrimes = 0;
for(int i = 0; i < primes.length; i++)
    sumOfPrimes += primes[i];
```

这种 `for` 循环结构很有特色，会经常见到。Java 还支持遍历句法，前面已经介绍过。上述求和代码可以改写成下述简洁的代码：

```
for(int p : primes) sumOfPrimes += p;
```

4. 复制数组

所有数组类型都实现了 `Cloneable` 接口，任何数组都能调用 `clone()` 方法复制自己。注意，返回值必须校正成适当的数组类型。不过，在数组上调用 `clone()` 方法不会抛出

CloneNotSupportedException 异常：

```
int[] data = { 1, 2, 3 };
int[] copy = (int[]) data.clone();
```

clone() 方法执行的是浅复制。如果数组的元素是引用类型，那么只复制引用，而不复制引用的对象。因为这种复制是浅复制，所以任何数组都能被复制，就算元素类型没有实现 Cloneable 接口也行。

不过，有时只想把一个现有数组中的元素复制到另一个现有数组中。System.arraycopy() 方法的目的就是高效完成这种操作。你可以假定 Java 虚拟机实现会在底层硬件中使用高速块复制操作执行这个方法。

arraycopy() 方法的作用简单明了，但使用起来有些难度，因为要记住五个参数。第一个参数是想从中复制元素的源数组；第二个参数是源数组中起始元素的索引；第三个参数是目标数组；第四个参数是目标索引；第五个参数是要复制的元素数量。

就算重叠复制同一个数组，arraycopy() 方法也能正确运行。例如，把数组 a 中索引为 0 的元素删除后，想把索引为 1 到 n 的元素向左移，把索引变成 0 到 n-1，可以这么做：

```
System.arraycopy(a, 1, a, 0, n);
```

5. 数组的实用方法

java.util.Arrays 类中包含很多处理数组的静态实用方法。这些方法中大多数都高度重载，有针对各种基本类型数组的版本，也有针对对象数组的版本。排序和搜索数组时，sort() 和 binarySearch() 方法特别有用。equals() 方法用于比较两个数组的内容。如果想把数组的内容转换成一个字符串，例如用于调试或记录日志，Arrays.toString() 方法很有用。

Arrays 类中还包含能正确处理多维数组的方法，例如 deepEquals()、deepHashCode() 和 deepToString()。

2.8.4　多维数组

前面已经见过，数组类型的写法是在元素类型后面加一对方括号。char 类型元素组成的数组是 char[] 类型，由 char[] 类型元素组成的数组是 char[][] 类型。如果数组的元素也是数组，我们说这个数组是多维数组。要想使用多维数组，需要了解一些其他细节。

假如想使用多维数组表示乘法表：

```
int[][] products;      // 乘法表
```

每对方括号表示一个维度，所以这是个二维数组。若想访问这个二维数组中的某个 int 元素，必须指定两个索引值，一个维度一个。假设这个数组确实被初始化成一个乘法

表，那么元素中存储的 int 值就是两个索引的乘积。也就是说，products[2][4] 的值是 8，products[3][7] 的值是 21。

创建多维数组要使用 new 关键字，而且要指定每个维度中数组的大小。例如：

```
int[][] products = new int[10][10];
```

在某些语言中，会把这样的数组创建成包含 100 个 int 值的数组，但 Java 不会这样处理。这行代码会做三件事。

- 声明一个名为 products 的变量，保存一个由 int[] 类型数组组成的数组。
- 创建一个有 10 个元素的数组，保存 10 个 int[] 类型的数组。
- 再创建 10 个数组，每个都由 10 个 int 类型的元素组成。然后把这 10 个新数组指定为前一步创建的数组的元素。这 10 个新数组中的每一个 int 类型元素的默认值都是 0。

换种方式说，前面的单行代码等效于下述代码：

```
int[][] products = new int[10][];  // 保存10个int[]类型值的数组
for(int i = 0; i < 10; i++)        // 循环10次……
    products[i] = new int[10];     // ……创建10个数组
```

new 关键字会自动执行这些额外的初始化操作。超过两个维度的数组也是一样：

```
float[][][] globalTemperatureData = new float[360][180][100];
```

使用 new 关键字创建多维数组时，无需指定所有维度的大小，只要为最左边的几个维度指定大小就行。例如，下面两行代码都是合法的：

```
float[][][] globalTemperatureData = new float[360][][];
float[][][] globalTemperatureData = new float[360][180][];
```

第一行代码创建一个一维数组，元素是 float[][] 类型。第二行代码创建一个二维数组，元素是 float[] 类型。不过，如果只为数组的部分维度指定大小，这些维度必须位于最左边。下述代码是不合法的：

```
float[][][] globalTemperatureData = new float[360][][100]; // 错误！
float[][][] globalTemperatureData = new float[][180][100]; // 错误！
```

和一维数组一样，多维数组也能使用数组初始化程序初始化，使用嵌套的花括号把数组嵌套在数组中即可。例如，可以像下面这样声明、创建并初始化一个 5×5 乘法表：

```
int[][] products = { {0, 0, 0, 0, 0},
                     {0, 1, 2, 3, 4},
                     {0, 2, 4, 6, 8},
                     {0, 3, 6, 9, 12},
                     {0, 4, 8, 12, 16} };
```

如果不想声明变量就使用多维数组，可以使用匿名初始化程序句法：

```
boolean response = bilingualQuestion(question, new String[][] {
                                                { "Yes", "No" },
                                                { "Oui", "Non" }});
```

使用 new 关键字创建多维数组时，往往最好只使用矩形数组，即每个维度的数组大小相同。

2.9　引用类型

至此，我们已经介绍了数组、类和对象，接下来可以介绍更一般的引用类型了。类和数组是 Java 五种引用类型中的两种。前面已经介绍了类，第 3 章会全面详细地说明类和接口。枚举和注解这两种引用类型在第 4 章介绍。

本节不涉及任何引用类型的具体句法，而是说明引用类型的一般行为，还会说明引用类型和基本类型的区别。本节使用术语“对象”指代引用类型（包括数组）的值或实例。

2.9.1　引用类型与基本类型比较

引用类型和对象与基本类型和基本值有本质的区别。

- 八种基本类型由 Java 语言定义，程序员不能定义新基本类型。引用类型由用户定义，因此有无限多个。例如，程序可以定义一个名为 Point 的类，然后使用这个新定义类型的对象存储和处理笛卡儿坐标系中的 (x, y) 点。
- 基本类型表示单个值。引用类型是聚合类型（aggregate type），可以保存零个或多个基本值或对象。例如，我们假设的 Point 类可能存储了两个 double 类型的值，表示点的 x 和 y 坐标。char[] 和 Point[] 数组类型是聚合类型，因为它们保存一些 char 类型的基本值或 Point 对象。
- 基本类型需要一到八个字节的内存空间。把基本值存储到变量中，或者传入方法时，计算机会复制表示这个值的字节。而对象基本上需要更多的内存。创建对象时会在堆（heap）中动态分配内存，存储这个对象；如果不再需要使用这个对象了，存储它的内存会被自动垃圾回收。

> 把对象赋值给变量或传入方法时，不会复制表示这个对象的内存，而是把这个内存的引用存储在变量中或传入方法。

在 Java 中，引用完全不透明，引用的表示方式由 Java 运行时的实现细节决定。如果你是

C 程序员的话，完全可以把引用看作指针或内存地址。不过要记住，Java 程序无法使用任何方式处理引用。

与 C 和 C++ 中的指针不同的是，引用不能转换成整数，也不能把整数转换成引用，而且不能递增或递减。C 和 C++ 程序员还要注意，Java 不支持求地址运算符 `&`，也不支持解除引用运算符 `*` 和 `->`。

2.9.2 处理对象和引用副本

下述代码处理 `int` 类型基本值：

```
int x = 42;
int y = x;
```

执行这两行代码后，变量 `y` 中保存了变量 `x` 中所存值的一个副本。在 Java 虚拟机内部，这个 32 位整数 42 有两个独立的副本。

现在，想象一下把这段代码中的基本类型换成引用类型后再运行会发生什么：

```
Point p = new Point(1.0, 2.0);
Point q = p;
```

运行这段代码后，变量 `q` 中保存了一份变量 `p` 中所存引用的一个副本。在虚拟机中，仍然只有一个 `Point` 对象的副本，但是这个对象的引用有两个副本——这一点有重要的含义。假设上面两行代码的后面是下述代码：

```
System.out.println(p.x);  // 打印p的x坐标:1.0
q.x = 13.0;               // 现在,修改q的x坐标
System.out.println(p.x);  // 再次打印p.x,这次得到的值是13.0
```

因为变量 `p` 和 `q` 保存的引用指向同一个对象，所以两个变量都可以用来修改这个对象，而且一个变量中的改动在另一个变量中可见。数组也是一种对象，所以对数组来说也会发生同样的事，如下面的代码所示：

```
// greet保存一个数组的引用
char[] greet = { 'h','e','l','l','o' };
char[] cuss = greet;            // cuss保存的是同一个数组的引用
cuss[4] = '!';                  // 使用引用修改一个元素
System.out.println(greet);      // 打印“hell!”
```

把基本类型和引用类型的参数传入方法时也有类似的区别。假如有下面的方法：

```
void changePrimitive(int x) {
    while(x > 0) {
        System.out.println(x--);
    }
}
```

调用这个方法时，会把实参的副本传给形参 x。在这个方法的代码中，x 是循环计数器，向零递减。因为 x 是基本类型，所以这个方法有这个值的私有副本——这是完全合理的做法。

可是，如果把这个方法的参数改为引用类型，会发生什么呢?

```
void changeReference(Point p) {
    while(p.x > 0) {
        System.out.println(p.x--);
    }
}
```

调用这个方法时，传入的是一个 Point 对象引用的私有副本，然后使用这个引用修改对应的 Point 对象。例如，有下述代码：

```
Point q = new Point(3.0, 4.5); // 一个x坐标为3的点
changeReference(q);            // 打印3,2,1,而且修改了这个Point对象
System.out.println(q.x);       // 现在,q的x坐标是0!
```

调用 changeReference() 方法时，传入的是变量 q 中所存引用的副本。现在，变量 q 和方法的形参 p 保存的引用指向同一个对象。这个方法可以使用它的引用修改对象的内容。但是要注意，这个方法不能修改变量 q 的内容。也就是说，这个方法可以随意修改引用的 Point 对象，但不能改变变量 q 引用这个对象这一事实。

2.9.3 比较对象

我们已经介绍了基本类型和引用类型在赋值给变量、传入方法和复制时的显著区别。这两种类型在相等性比较时也有区别。相等运算符（==）比较基本值时，只测试两个值是否一样（即每一位的值都完全相同）。而 == 比较引用类型时，比较的是引用而不是真正的对象。也就是说，== 测试两个引用是否指向同一个对象，而不测试两个对象的内容是否相同。例如：

```
String letter = "o";
String s = "hello";              // 这两个String对象
String t = "hell" + letter;      // 保存的文本完全一样
if (s == t) System.out.println("equal"); // 但是,二者并不相等!

byte[] a = { 1, 2, 3 };
// 内容一样的副本
byte[] b = (byte[]) a.clone();
if (a == b) System.out.println("equal"); // 但是,二者并不相等!
```

对引用类型来说，有两种相等：引用相等和对象相等。一定要把这两种相等区分开。其中一种方式是，使用“相同”（identical）表示引用相等，使用“相等”（equal）表示对象的内容一样。若想测试两个不同的对象是否相等，可以在一个对象上调用 equals() 方法，然后把另一个对象传入这个方法：

```
String letter = "o";
String s = "hello";             // 这两个String对象
String t = "hell" + letter;     // 保存的文本完全一样
if (s.equals(t)) {              // equals()方法
    System.out.println("equal"); // 证实了这一点
}
```

所有对象都（从 Object 类）继承了 equals() 方法，但是默认的实现方式是使用 == 测试引用是否相同，而不测试内容是否相等。想比较对象是否相等的类可以自定义 equals() 方法。Point 类没自定义，但 String 类自定义了，如前面的例子所示。可以在数组上调用 equals() 方法，但作用和使用 == 运算符一样，因为数组始终继承默认的 equals() 方法，比较引用而不是数组的内容。比较数组是否相等可以使用 java.util.Arrays.equals() 实用方法。

2.9.4 装包和拆包转换

基本类型和引用类型的表现完全不同。有时需要把基本值当成对象，为此，Java 平台为每一种基本类型都提供了*包装类*。Boolean、Byte、Short、Character、Integer、Long、Float 和 Double 是不可变的最终类，每个实例只保存一个基本值。包装类一般在把基本值存储在集合中时使用，例如 java.util.List：

```
// 创建一个List集合
List numbers = new ArrayList();
// 存储一个包装类表示的基本值
numbers.add(new Integer(-1));
// 取出这个基本值
int i = ((Integer)numbers.get(0)).intValue();
```

Java 支持装包和拆包类型转换。装包转换把一个基本值转换成对应的包装对象，而拆包转换的作用相反。虽然可以通过校正显式指定装包和拆包转换，但没必要这么做，因为把值赋值给变量或传入方法时会自动执行这种转换。此外，如果把包装对象传给需要基本值的 Java 运算符或语句，也会自动执行拆包转换。因为 Java 能自动执行装包和拆包转换，所以这种语言特性一般叫作*自动装包*（autoboxing）。

下面是一些自动装包和拆包转换的示例：

```
Integer i = 0;    // 把int类型字面量0装包到Integer对象中
Number n = 0.0f; // 把float类型字面量装包到Float对象中,然后放大转换成Number类型
Integer i = 1;    // 这是装包转换
int j = i;        // i在这里拆包
i++;              // 拆包i,递增,再装包
Integer k = i+2; // 拆包i,再装包两数之和
i = null;
j = i;            // 这次拆包抛出NullPointerException异常
```

自动装包也把集合处理变得更简单了。下面这个示例，使用 Java 的*泛型*（4.2 节专门介绍这个语言特性）限制列表和其他集合中能存储什么类型的值：

```
List<Integer> numbers = new ArrayList<>(); // 创建一个由整数组成的列表
numbers.add(-1);                           // 把int类型的值装包到Integer对象中
int i = numbers.get(0);                    // 把Integer对象拆包成int类型
```

2.10　包和Java命名空间

包由一些具名的类、接口和其他引用类型组成，目的是把相关的类组织在一起，并为这些类定义命名空间。

Java 平台的核心类放在一些名称以 java 开头的包中。例如，Java 语言最基本的类在 java.lang 包中，各种实用类在 java.util 包中，输入输出类在 java.io 包中，网络类在 java.net 包中。有些包还包含子包，例如 java.lang.reflect 和 java.util.regex。甲骨文标准化的 Java 平台扩展一般在名称以 javax 开头的包中。有些扩展，例如 javax.swing 及其各种子包，后来集成到了核心平台中。最后，Java 平台还包含几个被认可的标准，这些包以标准制定方命名，例如 org.w3c 和 org.omg。

每个类都有两个名称：一个是简称，定义时指定；另一个是完全限定名称，其中包含所在包的名称。例如，String 类是 java.lang 包的一部分，因此它的完全限定名称是 java.lang.String。

本节说明如何把自己的类和接口放到包里，以及如何选择包名，避免和其他人的包名有冲突。然后说明如何有选择性地把类型名称或静态成员导入命名空间，避免每次使用类或接口都要输入包名。

2.10.1　声明包

若想指定类属于哪个包，要使用 package 声明。如果 Java 文件中有 package 关键字，必须是 Java 代码的第一个标记（即除了注释和空格之外的第一个标记）。package 关键字后面是包的名称和一个分号。例如，有个 Java 文件以下述指令开头：

```
package org.apache.commons.net;
```

那么，这个文件中定义的所有类都是 org.apache.commons.net 包的一部分。

如果 Java 文件中没有 package 指令，那么这个文件中定义的所有类都是一个默认的无名包的一部分。此时，类的限定名称和不限定名称相同。

> 包的名称有可能冲突，所以不要使用默认包。项目在增长的过程中越来越复杂，冲突几乎是不可避免的，所以最好从一开始就创建包。

2.10.2 全局唯一的包名

包的重要功能之一是划分 Java 命名空间，避免类名有冲突。例如，只能从包名上区分 `java.util.List` 和 `java.awt.List` 两个类。不过，因此包名本身就要独一无二。作为 Java 的开发方，甲骨文控制着所有以 `java`、`javax` 和 `sun` 开头的包名。

常用的命名方式之一是使用自己的域名，倒序排列各部分，作为包名的前缀。例如，Apache 项目开发了一个网络库，是 Apache Commons 项目的一部分。Commons 项目的网址是 http://commons.apache.org/，因此这个网络库的包名是 `org.apache.commons.net`。

注意，API 开发者以前也使用这种包命名规则。如果其他程序员要把你开发的类和其他未知类放在一起使用，你的包名就要具有全局唯一性。如果你开发了一个 Java 程序，但是不会发布任何类供他人使用，那么你就知道部署这个应用需要使用的所有类，因此无需担心无法预料的命名冲突。此时，可以选择一种自己用着方便的命名方式，而不用考虑全局唯一性。常见的做法之一是，使用程序的名称作为主包的名称（主包里可能还有子包）。

2.10.3 导入类型

默认情况下，在 Java 代码中引用类或接口时，必须使用类型的完全限定名称，即包含包名。如果编写的代码需要使用 `java.io` 包中的 `File` 类处理文件，必须把这个类写成 `java.io.File`。不过这个规则有三个例外：

- `java.lang` 包中的类型很重要也很常用，因此始终可以使用简称引用；
- `p.T` 类型中的代码可以使用简称引用 `p` 包中定义的其他类型；
- 已经使用 `import` 声明导入命名空间里的类型，可以使用简称引用。

前两个例外叫作“自动导入”。`java.lang` 包和当前包中的类型已经导入到命名空间里了，因此可以不加包名。输入不在 `java.lang` 包或当前包中的常用类型的包名，很快就会变得冗长乏味，因此要能显式地把其他包中的类型导入命名空间。这种操作通过 `import` 声明实现。

`import` 声明必须放在 Java 文件的开头，如果有 `package` 声明的话，要紧随其后，并且在任何类型定义之前。一个文件中能使用的 `import` 声明数量不限。`import` 声明应用于文件中的所有类型定义（但不应用于 `import` 声明中的类型）。

`import` 声明有两种格式。若想把单个类型导入命名空间，`import` 关键字后面是类型的名称和一个分号：

```
import java.io.File;    // 现在不用输入java.io.File了,输入File就行
```

这种格式叫“单个类型导入”声明。

`import` 声明的另一种格式是“按需类型导入”。在这种格式中，包名后面是 `.*` 字符，表

示使用这个包里的任何类型时都不用输入包名。因此，如果除了 File 类之外，还要使用 java.io 包中的其他几个类，可以导入整个包：

```
import java.io.*;  // java.io包中的所有类都可以使用简称
```

按需导入句法对子包无效。如果导入了 java.util 包，仍然必须使用完全限定名称 java.util.zip.ZipInputStream 引用这个类。

按需导入类型和一个一个导入包中的所有类型作用不一样。按需导入更像是使用单个类型导入句法把代码中真正用到的各种类型从包中导入命名空间，因此才叫“按需”导入——用到某个类型时才会将其导入。

命名冲突和遮盖

import 声明对 Java 编程极其重要。不过，可能会导致命名冲突。例如，java.util 和 java.awt 两个包中都有名为 List 的类型。

java.util.List 是常用的重要接口。java.awt 包中有很多客户端应用常用的重要类型，但 java.awt.List 已经作废了，不是这些重要类型的其中一个。在同一个 Java 文件中既导入 java.util.List 又导入 java.awt.List 是不合法的。下述单个类型导入声明会导致编译出错：

```
import java.util.List;
import java.awt.List;
```

使用按需类型导入句法导入这两个包是合法的：

```
import java.util.*; // 导入集合和其他实用类型
import java.awt.*;  // 导入字体,颜色和图形类型
```

可是，如果试图使用 List 类型会遇到困难。这个类型可以从两个包中的任何一个“按需”导入，只要试图使用未限定的类型名引用 List 就会导致编译出错。这种问题的解决方法是，明确指定所需的包名。

因为 java.util.List 比 java.awt.List 常用得多，所以可以在两个按需类型导入声明后使用单个类型导入声明指明从哪个包中导入 List：

```
import java.util.*;     // 导入集合和其他实用类型
import java.awt.*;      // 导入字体,颜色和图形类型
import java.util.List; // 与java.awt.List区分开
```

这样，使用 List 时指的是 java.util.List 接口。如果确实需要使用 java.awt.List 类，只要加上包名就行。除此之外，java.util 和 java.awt 之间没有命名冲突了，在不指定包名的情况下使用这两个包中的其他类型时，会“按需”将其导入。

2.10.4 导入静态成员

除了类型之外，还可以使用关键字 import static 导入类型中的静态成员（静态成员在第 3 章说明。如果不熟悉这个概念，可以稍后再读这一节）。和类型导入声明一样，静态成员导入声明也有两种格式：单个静态成员导入和按需静态成员导入。假如你在编写一个基于文本的程序，要向 System.out 输出大量内容，那么可以使用下述单个静态成员导入声明减少输入的代码量：

```
import static java.lang.System.out;
```

加入这个导入声明后，可以用 out.println() 代替 System.out.println()。又假如你编写的一个程序要使用 Math 类中的很多三角函数和其他函数。在这种明显要大量使用数字处理方法的程序中，重复输入类名“Math”不会让代码的思路更清晰，反而会起到反作用。遇到这种情况，或许应该按需导入静态成员：

```
import static java.lang.Math.*
```

加入这个导入声明后，可以编写 sqrt(abs(sin(x))) 这样简洁的表达式，而不用在每个静态方法前都加上类名 Math。

import static 声明另一个重要的作用是把常量导入代码，尤其适合导入枚举类型（参见第 4 章）。假如你想在自己编写的代码中使用下述枚举类型中的值：

```
package climate.temperate;
enum Seasons { WINTER, SPRING, SUMMER, AUTUMN };
```

那么，可以导入 climate.temperate.Seasons，然后在常量前加上类型名，例如 Seasons.SPRING。如果想编写更简洁的代码，可以导入这个枚举类型中的值：

```
import static climate.temperate.Seasons.*;
```

使用静态成员导入声明导入常量一般来说比实现定义常量的接口更好。

静态成员导入和重载的方法

静态成员导入声明导入的是“名称”，而不是以这个名称命名的某个具体成员。因为 Java 允许重载方法，也允许类型中的字段和方法同名，所以单个静态成员导入声明可能会导入多个成员。例如下述代码：

```
import static java.util.Arrays.sort;
```

这个声明把名称“sort”导入命名空间，而没有导入 java.util.Arrays 里定义的 19 个 sort() 方法中的任何一个。如果使用导入的名称 sort 调用方法，编译器会根据方法的参数类型决定调用哪个方法。

从两个或多个不同的类型中导入同名的静态方法也是合法的，只要方法的签名不同就行。下面举个例子：

```
import static java.util.Arrays.sort;
import static java.util.Collections.sort;
```

你可能觉得上述代码会导致句法错误，其实不然，因为 `Collections` 类中定义的 `sort()` 方法和 `Arrays` 类中定义的所有 `sort()` 方法签名都不一样。在代码中使用“sort”这个名称时，编译器会根据参数的类型决定使用这 21 个方法中的哪一个。

2.11 Java文件的结构

本章从小到大说明了 Java 句法的元素，先介绍了单个字符和标记，然后介绍了运算符、表达式、语句和方法，最后介绍了类和包。从实际使用的角度出发，最常使用的 Java 程序结构单元是 Java 文件。Java 文件是 Java 编译器能编译的 Java 代码的最小单元。一个 Java 文件中包含以下内容：

- 一个可选的 `package` 指令；
- 零个或多个 `import` 或 `import static` 指令；
- 一个或多个类型定义。

当然，这些元素之间可以穿插注释，但必须是这种顺序。这就是 Java 文件中的全部内容了。所有 Java 语句都必须放在方法中（不含 `package` 和 `import` 指令，它们不是真正的语句），而所有方法都要放在类型定义中。

Java 文件还有一些其他重要的限制。首先，一个文件中最多只能有一个声明为 `public` 的顶层类。`public` 类的目的是供其他包中的类使用。但是，在一个类中，声明为 `public` 的嵌套类或内部类数量不限。第 3 章会详细介绍 `public` 修饰符和嵌套类。

第二个限制涉及到 Java 文件的文件名。如果 Java 文件中有一个 `public` 类，那么这个文件的名称必须和这个类的名称相同，然后再加上扩展名 .java。因此，如果 `Point` 定义为 `public` 类，那么它的源码要放在名为 Point.java 的文件中。不管类是否为 `public`，一个文件中只定义一个类，并使用类名命名文件，是良好的编程习惯。

编译 Java 文件时，其中定义的各个类会编译到独自的类文件中；类文件中是 Java 字节码，由 Java 虚拟机解释执行。类文件的名称和其中定义的类名相同，扩展名为 .class。因此，如果 Point.java 文件中定义了一个名为 `Point` 的类，那么，Java 编译器编译后得到的文件名为 Point.class。在大多数系统中，类文件都存储在包名对应的目录里。因此，`com.davidflanagan.examples.Point` 类在 com/davidflanagan/examples/Point.class 文件中定义。

Java 解释器知道标准系统类的类文件存储的位置，需要时会加载这些类文件。解释器运

行程序时，如果需要使用名为 `com.davidflanagan.examples.Point` 的类，它知道这个类的代码存储在名为 com/davidflanagan/examples/ 的目录中；默认情况下，解释器会在当前目录中寻找这个子文件夹。如果想告诉解释器在当前目录之外的位置寻找，调用解释器时必须使用 `-classpath` 选项，或者设定 `CLASSPATH` 环境变量。详情参见第 8 章对 Java 解释器（java）的说明。

2.12　定义并运行Java程序

Java 程序包含一系列相互作用的类定义，但不是每个 Java 类或 Java 文件都能当成程序。若想创建程序，必须在一个类中定义一个特殊的方法，签名如下：

```
public static void main(String[] args)
```

`main()` 方法是程序的主要入口，Java 解释器从这里开始运行。这个方法的参数是一个字符串数组，没有返回值。`main()` 方法返回后，Java 解释器也就退出了（除非 `main()` 方法创建了其他线程，此时，解释器会等到所有线程都结束后才会退出）。

Java 程序通过 Java 解释器（java）运行，并且要指定 `main()` 方法所在类的完全限定名称。注意，指定的是类名，而不是包含类的类文件名。命令行中指定的其他参数会传给 `main()` 方法的 `String[]` 参数。可能还要指定 `-classpath`（或 `-cp`）选项，告诉解释器在哪里寻找程序所需的类。例如，在下述命令中：

```
java -classpath /opt/Jude com.davidflanagan.jude.Jude datafile.jude
```

`java` 是运行 Java 解释器的命令；`-classpath /usr/local/Jude` 告诉解释器在哪里寻找类文件；`com.davidflanagan.jude.Jude` 是要运行的程序名（即定义 `main()` 方法的类名）；datafile.jude 是一个字符串，作为字符串数组的一个元素，传给 `main()` 方法。

运行程序有一种简单的方式。如果把程序及其所有辅助类都正确打包到一个 Java 档案（Java archive，JAR）文件中，那么只指定 JAR 文件的名称就可以运行这个程序。下面这个示例展示如何运行 Censum 垃圾回收日志分析程序：

```
java -jar /usr/local/Censum/censum.jar
```

在某些操作系统中，JAR 文件能自动执行。在这些系统中，可以直接运行：

```
% /usr/local/Censum/censum.jar
```

第 13 章会详细说明如何执行 Java 程序。

2.13　小结

本章介绍了 Java 语言的基本句法。编程语言的句法之间环环相扣，如果现在没有完全理解 Java 语言的全部句法，也没有关系，不管是人类还是计算机，都要通过实践才能精通任何一门语言。

还有一点要注意，有些句法比其他句法更常用。例如，`strictfp` 和 `assert` 两个关键字几乎从不使用。不要试图掌握 Java 句法的所有细节，最好先熟悉 Java 的核心概念，然后再回过头学习还不理解的句法细节。知道这一点之后，开始读下一章吧。下一章介绍对 Java 来说十分重要的类和对象，以及 Java 实现面向对象编程的基本方式。

第 3 章

Java面向对象编程

介绍 Java 基本句法之后，可以开始介绍 Java 面向对象编程了。所有 Java 程序都使用对象，对象的类型由*类*或*接口*定义。每个 Java 程序都定义成类，而复杂的程序会定义很多类和接口。本章说明如何定义新类，以及如何使用类进行面向对象编程。本章还会介绍接口的概念，但接口和 Java 的类型系统的详细介绍将在第 4 章进行。

> 不过，如果你有面向对象编程的经验，要小心。“面向对象”在不同的语言中有不同的含义。不要认为 Java 对面向对象的实现和你最喜欢的面向对象语言一样（C++ 和 Python 程序员尤其要注意）。

这一章的内容很多，下面先简要介绍一些基本概念。

3.1 类简介

类是 Java 程序最基本的元素结构。编写 Java 代码不可能不定义类。所有 Java 语句都在类中，而且所有方法都在类中实现。

3.1.1 面向对象的基本概念

下面是两个重要的概念。

- *类*

  类由一些保存值的数据字段和处理这些值的方法组成。类定义一种新的引用类型，例如

第 2 章定义的 Point 类型。

Point 类定义的类型能表示所有二维点。

- *对象*

  对象是类的*实例*。

Point 对象是这个类型的一个值，即表示一个二维点。

对象一般通过*实例化*类创建，方法是使用 new 关键字并调用构造方法，如下所示：

```
Point p = new Point(1.0, 2.0);
```

构造方法将在 3.3 节介绍。

一个类的定义包含一个*签名*和一个*主体*。类的签名定义类的名称，可能还会指定其他重要信息。类的主体是一些放在花括号里的*成员*。类的成员一般包含字段和方法，也可以包含构造方法、初始化程序和嵌套类型。

成员可以是*静态*的，也可以是非静态的。静态成员属于类本身，而非静态成员关联在类的实例上（参见 3.2 节）。

> 常见的成员有四种：类字段、类方法、实例字段和实例方法。Java 的主要工作就是与这些成员交互。

类的签名可能会声明它*扩展*自其他类。被扩展的类叫作*超类*，扩展其他类的类叫作*子类*。子类*继承*超类的成员，而且可以声明新成员，或者使用新的实现*覆盖*继承的方法。

类的成员可以使用*访问修饰符* public、protected 或 private。[1] 这些修饰符指定成员在使用方和子类中是否可见以及能否访问。类通过这种方式控制对非公开 API 成员的访问。隐藏成员是一种面向对象设计技术，叫作*数据封装*（data encapsulation），3.5 节会介绍。

3.1.2 其他引用类型

类的签名可能还会声明类*实现*了一个或多个接口。*接口*是一种类似于类的引用类型，其中定义了方法签名，但一般没有实现方法的方法主体。

不过，从 Java 8 开始，接口可以使用关键字 default 指明其中的方法是可选的。如果方法是可选的，接口文件必须包含默认的实现（因此才选用 default 这个关键词）；所有实现

注 1：稍后会见到默认的可见性，即在包中可见。

这个接口的类，如果没有实现可选的方法，就使用接口中默认的实现。

实现接口的类必须为接口的非默认方法提供主体。实现某个接口的类的实例，也是这个接口类型的实例。

类和接口是 Java 定义的五种基本引用类型中最重要的两个。另外三个基本引用类型是数组、枚举类型和注解类型（通常直接叫“注解”）。第 2 章已经介绍过数组。枚举是特殊的类，注解是特殊的接口——第 4 章会介绍这两种类型，还会全面说明接口。

3.1.3 定义类的句法

最简单的类定义方式是在关键字 `class` 后面放上类的名称，然后在花括号中放一些类的成员。`class` 关键字前面可以放修饰符关键字或注解。如果类扩展其他类，类名后面要加上 `extends` 关键字和要扩展的类名。如果类实现一个或多个接口，类名或 `extends` 子句之后要加上 `implements` 关键字和用逗号分隔的接口名。例如：

```
public class Integer extends Number implements Serializable, Comparable {
    // 这里是类的成员
}
```

定义泛型类时还可以指定类型参数和通配符（参见第 4 章）。

类声明可以包含修饰符关键字。除访问控制修饰符（`public`、`protected` 等）之外，还可以使用：

- `abstract`

  `abstract` 修饰的类未完全实现，不能实例化。只要类中有 `abstract` 修饰的方法，这个类就必须使用 `abstract` 声明。抽象类在 3.6 节介绍。

- `final`

  `final` 修饰符指明这个类无法被扩展。类不能同时声明为 `abstract` 和 `final`。

- `strictfp`

  如果类声明为 `strictfp`，那么其中所有的方法都声明为 `strictfp`。这个修饰符极少使用。

3.2 字段和方法

类可以看成是由一些数据（也叫状态）和操作这些状态的代码组成的。数据存储在字段中，操作数据的代码则组织在方法中。

本节介绍两种最重要的类成员：字段和方法。字段和方法有两种不同的类型：关联在类自身上的类成员（也叫静态成员），关联在类的单个实例（即对象）身上的实例成员。因此，

成员分为四类：

- 类字段
- 类方法
- 实例字段
- 实例方法

示例 3-1 定义了一个简单的类 `Circle`，包含所有这四种成员类型。

示例 3-1：一个简单的类及其成员

```
public class Circle {
  // 类字段
  public static final double PI= 3.14159;  // 有用的常量

  // 类方法:基于参数计算得到一个值
  public static double radiansToDegrees(double radians) {
    return radians * 180 / PI;
  }

  // 实例字段
  public double r;               // 圆的半径

  // 两个实例方法:处理对象的实例字段
  public double area() {         // 计算圆的面积
    return PI * r * r;
  }

  public double circumference() { // 计算圆的周长

    return 2 * PI * r;
  }
}
```

一般来说公开 `r` 字段并不好，最好把 `r` 声明为私有字段，然后提供 `radius()` 方法，获取它的值。原因在 3.5 节说明。现在，我们使用公开字段只是为了演示如何处理实例字段。

随后的几节说明这四种成员。首先，介绍声明字段的句法。声明方法的句法在 3.5 节介绍。

3.2.1 声明字段的句法

声明字段的句法和声明局部变量的句法很像（参见第 2 章），不过声明字段时还可以使用修饰符。最简单的字段声明包含字段类型和字段名。类型前面可以放零个或多个修饰符关键字或注解，名称后面可以跟着一个等号和初始化表达式，提供字段的初始值。如果两个或多个字段的类型和修饰符都相同，那么可以把一些用逗号分隔的字段名和初始化表达式

放在类型后面。如下是一些有效的字段声明：

```
int x = 1;
private String name;
public static final int DAYS_PER_WEEK = 7;
String[] daynames = new String[DAYS_PER_WEEK];
private int a = 17, b = 37, c = 53;
```

字段的修饰符由零个或多个下述关键字组成。

- `public`、`protected`、`private`

  这些访问控制修饰符指明字段是否能在定义它的类之外使用，以及能在何处使用。

- `static`

  如果使用，这个修饰符指明字段关联在定义它的类自身上，而不是类的实例身上。

- `final`

  这个修饰符指明，字段一旦初始化，其值就不能改变。如果字段同时使用 `static` 和 `final` 修饰，那么这个字段就是编译时常量，javac 会将其内联化。`final` 修饰的字段也可以用来创建实例不可变的类。

- `transient`

  这个修饰符指明字段不是对象持久状态的一部分，无需跟对象的其他内容一起序列化。

- `volatile`

  这个修饰符指明字段有额外的语义，可被两个或多个线程同时使用。`volatile` 修饰符的意思是，字段的值必须始终从主存储器中读取和释放，不能被线程缓存（在寄存器或 CPU 缓存中）。详情参见第 6 章。

3.2.2　类字段

*类字段*关联在定义它的类身上，而不是类的实例身上。下面这行代码声明一个类字段：

```
public static final double PI = 3.14159;
```

这行代码声明了一个字段，类型为 `double`，名称为 `PI`，并且把值设为 3.14159。

`static` 修饰符表明这个字段是类字段。因为使用了 `static` 修饰符，所以类字段有时也叫静态字段。`final` 修饰符表明这个字段的值不会改变。因为字段 `PI` 表示一个常量，而且声明时加上了 `final`，所以无法修改它的值。在 Java（以及很多其他语言）中，习惯使用大写字母命名常量，因此这个字段的名称是 `PI`，而不是 `pi`。类字段经常用来定义常量，也就是说，`static` 和 `final` 修饰符经常放在一起使用。然而，并不是所有类字段都是常量，因此字段可以声明为 `static` 但不声明为 `final`。

公开的静态字段要尽量声明为 `final`，因为多个线程都能修改字段的值，会导致极难调试的行为。

公开的静态字段其实就是全局变量。不过，类字段的名称会被定义它的类名限定，因此，如果不同的模块定义了同名的全局变量，Java 不会出现其他语言遇到的名称冲突问题。

关于静态字段，有个重点要理解，即字段的值只有一个副本。字段关联在类自身上，而不是类的实例身上。看一下 `Circle` 类中的各个方法，它们都使用了同一个字段。在 `Circle` 类内部，可以直接使用 `PI` 引用这个字段。但是在类的外部，既要使用类名也要使用字段名，这样才能引用这个独一无二的字段。`Circle` 类外部的方法要使用 `Circle.PI` 才能访问这个字段。

3.2.3 类方法

和类字段一样，*类方法*也使用 `static` 修饰符声明：

```
public static double radiansToDegrees(double rads) {
  return rads * 180 / PI;
}
```

上述代码声明了一个类方法，名为 `radiansToDegrees()`。这个方法只有一个参数，类型为 `double`，而且会返回一个 `double` 类型的值。

和类字段一样，类方法也关联在类身上，而不是对象身上。在类的外部调用类方法时，既要指定类名也要指定方法名。例如：

```
// 2.0弧度等于多少角度?
double d = Circle.radiansToDegrees(2.0);
```

如果想在定义类方法的类中调用类方法，则不用指定类名。还可以使用静态成员导入声明，减少输入的代码量（参见第 2 章）。

注意，`Circle.radiansToDegrees()` 方法的主体使用了类字段 `PI`。类方法可以使用所在类（或其他类）中的任何类字段和类方法。

类方法不能使用任何实例字段或实例方法，因为类方法不关联在类的实例身上。也就是说，虽然 `radiansToDegrees()` 方法在 `Circle` 类中定义，但它不能使用 `Circle` 对象的任何实例成员。

可以这样理解：在任何实例中，总有一个 this 引用指向当前对象，但类方法不关联在具体的实例身上，所以没有 this 引用，因此不能访问实例字段。

前面说过，类字段其实就是全局变量。类似地，类方法是全局方法，或全局函数。虽然 radiansToDegrees() 方法不处理 Circle 对象，但还是在 Circle 类中定义，因为它是一个实用方法，处理圆时有时会用到，因此可以把它和 Circle 类的其他功能放在一起。

3.2.4 实例字段

声明时没使用 static 修饰符的字段是实例字段：

```
public double r;    // 圆的半径
```

实例字段关联在类的实例身上，所以创建的每个 Circle 对象都有自己的一个 double 类型 r 字段副本。在这个例子中，r 表示某个圆的半径。每个 Circle 对象的半径和其他所有 Circle 对象的都不同。

在类定义内部，实例字段只通过名称引用。在实例方法 circumference() 的主体中有一个例子。在类外部，实例字段的名称前面必须加上包含这个字段的对象的引用。例如，如果变量 c 保存的是一个 Circle 对象的引用，那么可以使用表达式 c.r 引用这个圆的半径：

```
Circle c = new Circle(); // 创建一个Circle对象,把引用存储在c中
c.r = 2.0;               // 把一个值赋值给实例字段r
Circle d = new Circle(); // 再创建一个Circle对象
d.r = c.r * 2;           // 让这个圆是前一个的两倍大
```

实例字段是面向对象编程的关键。实例字段保存对象的状态，实例字段的值把两个对象区分开来。

3.2.5 实例方法

实例方法处理类的具体实例（对象），只要声明方法时没使用 static 关键字，这个方法默认就是实例方法。

实例方法这个特性让面向对象编程开始变得有趣。示例 3-1 中定义的 Circle 类包含两个实例方法，area() 和 circumference()，分别计算指定 Circle 对象表示的圆的面积和周长。

若想在定义实例方法的类之外使用实例方法，必须在方法名前加上要处理的实例引用。例如：

```
// 创建一个Circle对象,存储在变量c中
Circle c = new Circle();
c.r = 2.0;              // 设定这个对象的实例字段
double a = c.area();    // 调用这个对象的实例方法
```

> 这就是叫面向对象编程的原因，这里对象是重点，而不是函数调用。

在实例方法内部，可以自然地访问属于调用这个方法的对象的实例字段。前面说过，经常可以把对象理解为包含状态（通过对象的字段表示）和行为（处理状态的方法）的包（bundle）。

实现所有实例方法时都使用了一个隐式参数，方法签名里没显示这个参数。这个隐式参数是 `this`，它的值是调用这个方法的对象引用。在我们的例子中，是一个 `Circle` 对象。

> `area()` 和 `circumference()` 两个方法的主体都使用了类字段 `PI`。前面说过，类方法只能使用类字段和类方法，而不能使用实例字段或实例方法。实例方法没有这种限制，不管类中的成员有没有声明为 `static`，实例方法都可以使用。

3.2.6 this引用的工作方式

方法签名中不显示隐式参数 `this`，是因为往往用不到。只要 Java 方法在类中访问实例字段，都默认访问 `this` 参数指向的对象中的字段。实例方法调用同一个类中的其他实例方法时也一样，可以理解为“在当前对象上调用实例方法”。

不过，如果想明确表明方法访问的是自己的字段或方法，可以显式使用 `this` 关键字。例如，可以改写 `area()` 方法，显式使用 `this` 引用实例字段：

```
public double area() { return Circle.PI * this.r * this.r; }
```

上述代码还显式使用类名引用类字段 `PI`。在这样简单的方法中，一般无需如此明确。然而，遇到复杂情况时，在不强制要求使用 `this` 的地方使用 `this`，有时可以让代码的意图更明确。

不过，有些情况下必须使用 `this` 关键字。例如，如果方法的参数或方法中的局部变量和类中的某个字段同名，那么就必须使用 `this` 引用这个字段，因为只使用字段名的话，引用的是方法的参数或局部变量。

例如，可以把下述方法添加到 `Circle` 类中：

```
    public void setRadius(double r) {
      this.r = r;       // 把参数r的值赋值给字段this.r
                        // 注意,不能写成r = r
    }
```

有些开发者会谨慎选择方法的参数名，避免和字段名冲突，因此可以最大限度地少使用 this。

最后，注意，实例方法可以使用 this 关键字，但类方法不能使用。这是因为类方法不关联在单个对象身上。

3.3 创建和初始化对象

介绍字段和方法之后，接下来要介绍类的其他重要成员。具体而言，我们要介绍构造方法。构造方法是类成员，作用是初始化新建实例中的字段。

再看一下创建 Circle 对象的方式：

```
Circle c = new Circle();
```

这行代码的意思是，调用看起来有点儿像方法的东西创建一个新 Circle 实例。其实，Circle() 是一种*构造方法*，是类中的成员，和类同名，而且像方法一样，有主体。

构造方法的工作方式是这样的：new 运算符表明我们想创建类的一个新实例。首先，分配内存存储新建的对象实例；然后，调用构造方法的主体，并传入指定的参数；最后，构造方法使用这些参数执行初始化新对象所需的一切操作。

Java 中的每个类都至少有一个构造方法，其作用是执行初始化新对象所需的操作。示例 3-1 定义的 Circle 类没有显式定义构造方法，因此 javac 编译器自动为我们提供了一个构造方法（叫作默认构造方法）。这个构造方法没有参数，而且不执行任何特殊的初始化操作。

3.3.1 定义构造方法

可是 Circle 对象显然要做些初始化操作，下面就来定义一个构造方法。示例 3-2 重新定义了 Circle 类，包含一个构造方法，指定新建 Circle 对象的半径。借此机会，我们还把 r 字段改成了受保护的（禁止对象随意访问）。

示例 3-2：为 Circle 类定义一个构造方法

```
public class Circle {
    public static final double PI = 3.14159; // 常量
    // 实例字段,保存圆的半径
    protected double r;

    // 构造方法:初始化r字段
```

```
    public Circle(double r) { this.r = r; }

    // 实例方法:基于半径计算得到值
    public double circumference() { return 2 * PI * r; }
    public double area() { return PI * r*r; }
    public double radius() { return r; }
}
```

如果依赖编译器提供的默认构造方法，就要编写如下的代码显式初始化半径：

```
Circle c = new Circle();
c.r = 0.25;
```

添加上述构造方法后，初始化变成创建对象过程的一部分：

```
Circle c = new Circle(0.25);
```

下面是一些关于命名、声明和编写构造方法的基本注意事项。

- 构造方法的名称始终和类名一样。
- 声明构造方法时不指定返回值类型，连 void 都不用。
- 构造方法的主体初始化对象。可以把主体的作用想象为设定 this 引用的内容。
- 构造方法不能返回 this 或任何其他值。

3.3.2 定义多个构造方法

有时，根据遇到的情况，可能想在多个不同的方式中选择一个最便利的方式初始化对象。例如，我们可能想使用指定的值初始化圆的半径，或者使用一个合理的默认值初始化。为 Circle 类定义两个构造方法的方式如下：

```
public Circle() { r = 1.0; }
public Circle(double r) { this.r = r; }
```

Circle 类只有一个实例字段，由此并没有太多的初始化方式。不过在复杂的类中，经常会定义不同的构造方法。

只要构造方法的参数列表不同，为一个类定义多个构造方法完全是合法的。编译器会根据提供的参数数量和类型判断你想使用的是哪个构造方法。定义多个构造方法和方法重载的原理类似。

3.3.3 在一个构造方法中调用另一个构造方法

如果类有多个构造方法，会用到 this 关键字的一种特殊用法。在一个构造方法中可以使用 this 关键字调用同一个类中的另一个构造方法。因此，前面 Circle 类的两个构造方法可以改写成：

```
// 这是基本构造方法:初始化半径
public Circle(double r) { this.r = r; }
// 这个构造方法使用this()调用前一个构造方法
public Circle() { this(1.0); }
```

如果一些构造方法共用大量的初始化代码，这种技术是有用的，因为能避免代码重复。如果构造方法执行很多初始化操作，在这种复杂的情况下，这种技术十分有用。

使用 `this()` 时有个重大的限制：只能出现在构造方法的第一个语句中。但是，调用这个方法后，可以执行构造方法所需的任何其他初始化操作。这个限制的原因涉及自动调用超类的构造方法，本章后面会说明。

3.3.4　字段的默认值和初始化程序

类中的字段不一定要初始化。如果没有指定初始值，字段自动使用默认值初始化：`false`、`\u0000`、`0`、`0.0` 或 `null`。具体使用哪个值，根据字段的类型而定（详情参见表 2-1）。这些默认值由 Java 语言规范规定，实例字段和类字段都适用。

如果字段的默认值不适合字段，可以显式提供其他的初始值。例如：

```
public static final double PI = 3.14159;
public double r = 1.0;
```

字段声明不是任何方法的一部分。Java 编译器会自动为字段生成初始化代码，然后把这些代码放在类的所有构造方法中。这些初始化代码按照字段在源码中出现的顺序插入构造方法，因此，字段的初始化程序可以使用在其之前声明的任何字段的初始值。

例如下述代码片段是一个假设类，定义了一个构造方法和两个实例字段：

```
public class SampleClass {
  public int len = 10;
  public int[] table = new int[len];

  public SampleClass() {
    for(int i = 0; i < len; i++) table[i] = i;
  }

  // 类余下的内容省略了……
}
```

对这个例子来说，`javac` 生成的构造方法其实和下述代码等效：

```
public SampleClass() {
  len = 10;
  table = new int[len];
```

```
    for(int i = 0; i < len; i++) table[i] = i;
  }
```

如果某个构造方法的开头使用 this() 调用其他构造方法，那么字段的初始化代码不会出现在这个构造方法中。此时，初始化由 this() 调用的构造方法处理。

既然实例字段在构造方法中初始化，那么类字段在哪初始化呢？就算从不创建类的实例，类字段也关联在类身上。这意味着，类字段要在调用构造方法之前初始化。

为此，javac 会为每个类自动生成一个类初始化方法。类字段在这个方法的主体中初始化。这个方法只在首次使用类之前调用一次（经常是在 Java 虚拟机首次加载类时）。

和实例字段的初始化一样，类字段的初始化表达式按照类字段在源码中的顺序插入类初始化方法。因此，类字段的初始化表达式可以使用在其之前声明的类字段。类初始化方法是内部方法，对 Java 程序员不可见。在类文件中，它的名称是 <clinit>（例如，使用 javap 检查类文件时可以看到这个方法。第 13 章会详细介绍如何使用 javap 执行这项操作）。

初始化程序块

至此，我们知道对象可以通过字段的初始化表达式和构造方法中的任何代码初始化。类有一个类初始化方法，这个方法和构造方法不一样，不能像构造方法那样显式定义主体。不过，Java 允许编写用于初始化类字段的代码，所用的结构叫*静态初始化程序*。静态初始化程序由 static 关键字及随后的花括号中的代码块组成。在类定义中，静态初始化程序可以放在字段和方法定义能出现的任何位置。例如，下述代码为两个类字段执行一些重要的初始化操作：

```
// 我们可以使用三角函数画出圆的轮廓
// 不过,三角函数很慢,所以预先算出一些值
public class TrigCircle {
  // 这是静态查找表和各自的初始化程序
  private static final int NUMPTS = 500;
  private static double sines[] = new double[NUMPTS];
  private static double cosines[] = new double[NUMPTS];

  // 这是一个静态初始化程序,填充上述数组
  static {
    double x = 0.0;
    double delta_x = (Circle.PI/2)/(NUMPTS-1);
    for(int i = 0, x = 0.0; i < NUMPTS; i++, x += delta_x) {
      sines[i] = Math.sin(x);
      cosines[i] = Math.cos(x);
    }
  }
  // 类余下的内容省略了……
}
```

一个类可以有任意多个静态初始化程序。各个初始化程序块的主体会和所有静态字段的初始化表达式一起合并到类初始化方法中。静态初始化程序和类方法的相同点是，不能使用

this 关键字，也不能使用类中的任何实例字段或实例方法。

类还可以有*实例初始化程序*。实例初始化程序和静态初始化程序类似，不过初始化的是对象而不是类。一个类可以有任意多个实例初始化程序，而且实例初始化程序可以放在字段和方法定义能出现的任何位置。各个实例初始化程序的主体和所有实例字段初始化表达式一起，放在类中每个构造方法的开头。实例初始化程序的外观和静态初始化程序类似，不过不使用 static 关键字。也就是说，实例初始化程序只是放在花括号里的任意 Java 代码。

实例初始化程序可以初始化数组或其他需要复杂初始化操作的字段。实例初始化程序有时很有用，因为它们把初始化代码放在字段后面，而不是单独放在构造方法中。例如：

```
private static final int NUMPTS = 100;
private int[] data = new int[NUMPTS];
{ for(int i = 0; i < NUMPTS; i++) data[i] = i; }
```

不过，现实中很少使用实例初始化程序。

3.4 子类和继承

前面定义的 Circle 是个简单的类，只通过半径区分不同的圆。假设我们要同时使用大小和位置表示圆。例如，在笛卡儿平面中，圆心在 (0, 0)、半径为 1.0 的圆，与圆心在 (1, 2)、半径为 1.0 的圆不同。为此，需要一个新类，我们称其为 PlaneCircle。

我们想添加表示圆所在位置的功能，但不想失去 Circle 类的任何现有功能。为此，可以把 PlaneCircle 类定义为 Circle 类的子类，让 PlaneCircle 类继承超类 Circle 的字段和方法。通过定义子类或扩展超类向类中添加功能的能力，是面向对象编程范式的核心。

3.4.1 扩展类

示例 3-3 展示了如何把 PlaneCircle 类定义为 Circle 类的子类。

示例 3-3：扩展 Circle 类

```
public class PlaneCircle extends Circle {
  // 自动继承了Circle类的字段和方法，
  // 因此只要在这里编写新代码
  // 新实例字段,存储圆心的位置
  private final double cx, cy;

  // 新构造方法,用于初始化新字段
  // 使用特殊的句法调用构造方法Circle()
  public PlaneCircle(double r, double x, double y) {
    super(r);       // 调用超类的构造方法Circle()
    this.cx = x;    // 初始化实例字段cx
    this.cy = y;    // 初始化实例字段cy
  }
```

```
    public double getCentreX() {
      return cx;
    }

    public double getCentreY() {
      return cy;
    }

    // area()和circumference()方法继承自Circle类
    // 新实例方法,检查点是否在圆内
    // 注意,这个方法使用了继承的实例字段r
    public boolean isInside(double x, double y) {
      double dx = x - cx, dy = y - cy;             // 到圆心的距离
      double distance = Math.sqrt(dx*dx + dy*dy); // 勾股定理
      return (distance < r);                       // 返回true或false
    }
  }
```

注意示例 3-3 第一行中使用的 `extends` 关键字。这个关键字告诉 Java，`PlaneCircle` 类扩展 `Circle` 类（或者说是 `Circle` 类的子类），这意味着 `PlaneCircle` 类会继承 `Circle` 类的字段和方法。

> 有多种方式能表达新对象类型具有 `Circle` 的特征，而且有位置。这或许是最简单的方式，但不一定是最合适的方式，尤其是在大型系统中。

`isInside()` 方法的定义展示了字段继承：这个方法使用了字段 `r`（由 `Circle` 类定义），就像这个字段是在 `PlaneCircle` 中定义的一样。`PlaneCircle` 还继承了 `Circle` 的方法。因此，如果变量 `pc` 保存的值是一个 `PlaneCircle` 对象引用，那么可以编写如下代码：

```
double ratio = pc.circumference() / pc.area();
```

这么做就好像 `area()` 和 `circumference()` 两个方法是在 `PlaneCircle` 中定义的一样。

子类的另一个特性是，每个 `PlaneCircle` 对象都是完全合法的 `Circle` 对象。如果 `pc` 是一个 `PlaneCircle` 对象的引用，那么可以把这个引用赋值给 `Circle` 类型的变量，忽略它表示的位置：

```
// 位置在原点的单位圆
PlaneCircle pc = new PlaneCircle(1.0, 0.0, 0.0);
Circle c = pc;     // 无需校正,赋值给Circle类型的变量
```

把 `PlaneCircle` 对象赋值给 `Circle` 类型的变量时无需校正。第 2 章说过，这种转换完全合法。`Circle` 类型的变量 `c` 中保存的值仍然是有效的 `PlaneCircle` 对象，但编译器不确定这

一点，因此不校正无法反向（缩小）转换：

```
// 缩小转换需要校正(虚拟机还要做运行时检查)
PlaneCircle pc2 = (PlaneCircle) c;
boolean origininside = ((PlaneCircle) c).isInside(0.0, 0.0);
```

4.5 节介绍编译时和运行时对象类型的区别时会详细说明这两种转换之间的不同。

`final`类

如果声明类时使用了 `final` 修饰符，那么这个类无法被扩展或定义子类。`java.lang.String` 是 `final` 类的一个示例。把类声明为 `final` 可以避免不需要的类扩展：在 `String` 对象上调用方法时，就算 `String` 类来自某个未知的外部源，你也知道这个方法是在 `String` 类中定义的。

3.4.2　超类、对象和类层次结构

在这个示例中，`PlaneCircle` 是 `Circle` 的子类，也可以说 `Circle` 是 `PlaneCircle` 的超类。类的超类在 `extends` 子句中指定：

```
public class PlaneCircle extends Circle { ... }
```

你定义的每个类都有超类。如果没使用 `extends` 子句指定超类，那么超类是 `java.lang.Object`。`Object` 是特殊的类，原因有如下两个：

- 它是 Java 中唯一一个没有超类的类；
- 所有 Java 类都从 `Object` 类中继承方法。

因为每个类（除了 `Object` 类）都有超类，所以 Java 中的类组成一个类层次结构。这个体系可以使用一个根为 `Object` 类的树状图表示。

> `Object` 类没有超类，而且其他每个类都只有一个超类。子类扩展的超类不能超过一个。第 4 章会详细说明如何实现类似的效果。

图 3-1 展示的是类层次结构的一部分，包含我们定义的 `Circle` 和 `PlaneCircle` 类，以及 Java API 中的一些标准类。

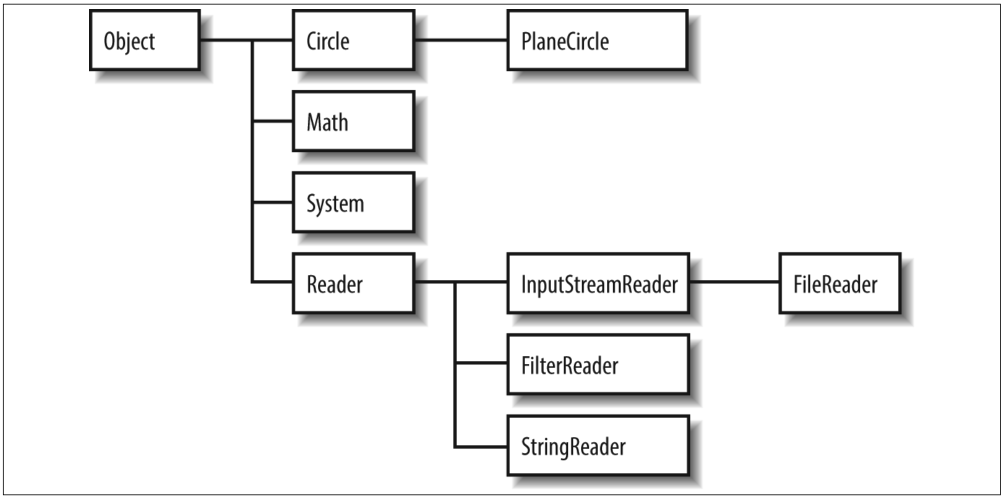


图 3-1：类层次结构图

3.4.3 子类的构造方法

再看一下示例 3-3 中的 PlaneCircle() 构造方法：

```
public PlaneCircle(double r, double x, double y) {
  super(r);       // 调用超类的构造方法Circle()
  this.cx = x;    // 初始化实例字段cx
  this.cy = y;    // 初始化实例字段cy
}
```

虽然这个构造方法显式初始化了 PlaneCircle 类中新定义的字段 cx 和 cy，但仍使用超类的 Circle() 构造方法初始化继承的字段。为了调用超类的构造方法，这个构造方法调用了 super() 方法。

super 是 Java 的保留字。它的用法之一是，在子类的构造方法中调用超类的构造方法。这种用法和在一个构造方法中使用 this() 调用同一个类中的其他构造方法类似。使用 super() 调用构造方法和使用 this() 调用构造方法有同样的限制：

- 只能在构造方法中像这样使用 super()；
- 必须在构造方法的第一个语句中调用超类的构造方法，甚至要放在局部变量声明之前。

传给 super() 的实参必须与超类构造方法的形参匹配。如果超类定义了多个构造方法，那么 super() 可以调用其中任何一个，具体是哪个，由传入的参数决定。

3.4.4 构造方法链和默认构造方法

创建类的实例时，Java 保证一定会调用这个类的构造方法；创建任何子类的实例时，Java

还保证一定会调用超类的构造方法。为了保证第二点，Java 必须确保每个构造方法都会调用超类的构造方法。

因此，如果构造方法的第一个语句没有使用 `this()` 或 `super()` 显式调用另一个构造方法，`javac` 编译器会插入 `super()`（即调用超类的构造方法，而且不传入参数）。如果超类没有无需参数的可见构造方法，这种隐式调用会导致编译出错。

以 `PlaneCircle` 类为例，创建这个类的新实例时会发生下述事情：

- 首先，调用 `PlaneCircle` 类的构造方法；
- 这个构造方法显示调用了 `super(r)`，调用 `Circle` 类的一个构造方法；
- `Circle()` 构造方法会隐式调用 `super()`，调用 `Circle` 的超类 `Object` 的构造方法（`Object` 只有一个构造方法）；
- 此时，到达层次结构的顶端了，接下来开始运行构造方法；
- 首先运行 `Object` 构造方法的主体；
- 返回后，再运行 `Circle()` 构造方法的主体；
- 最后，对 `super(r)` 的调用返回后，接着执行 `PlaneCircle()` 构造方法中余下的语句。

这个过程表明，构造方法链在一起调用；只要创建对象，就会调用一系列构造方法，从子类到超类，一直向上，直到类层次结构的顶端 `Object` 类为止。因为超类的构造方法始终在子类的构造方法的第一个语句中调用，所以 `Object` 类的构造方法的主体始终最先运行，然后运行 `Object` 的子类的构造方法，就这样沿着类层次结构一直向下，直到实例化的那个类为止。

调用构造方法时，超类中的字段也会被初始化。

默认构造方法

前面对构造方法链的说明漏了一点。如果构造方法没有调用超类的构造方法，Java 会隐式调用。那么，如果类没有声明构造方法呢？此时，Java 会为类隐式添加一个构造方法。这个默认的构造方法什么也不做，只是调用超类的构造方法。

例如，如果没为 `PlaneCircle` 类声明构造方法，那么 Java 会隐式插入下述构造方法：

```
public PlaneCircle() { super(); }
```

如果超类 `Circle` 没有声明无参数的构造方法，那么在这个自动插入 `PlaneCircle()` 类的默认构造方法中调用 `super()` 会导致编译出错。一般来说，如果类没有定义无参数的构造方法，那么它的所有子类必须定义显式调用超类构造方法的构造方法，而且要传入所需的参数。

如果类没有定义任何构造方法，默认会为其提供一个无参数的构造方法。声明为 public 的类，提供的构造方法也声明为 public。提供给其他类的默认构造方法则不使用任何可见性修饰符，这些构造方法具有默认的可见性。（本章后面会说明指定可见性的方式。）

如果创建的 public 类不能公开实例化，就应该至少声明一个非 public 的构造方法，以此避免插入默认的 public 构造方法。从来不会实例化的类（例如 java.lang.Math 或 java.lang.System），应该定义一个 private 构造方法。这种构造方法不能在类外部调用，但可以避免自动插入默认的构造方法。

3.4.5 遮盖超类的字段

假如 PlaneCircle 类需要知道圆心到原点 (0, 0) 的距离，我们可以再添加一个实例字段保存这个值：

```
public double r;
```

在构造方法中添加下述代码可以算出这个字段的值：

```
this.r = Math.sqrt(cx*cx + cy*cy); // 勾股定理
```

但是等一下，这个新添加的字段 r 和超类 Circle 中表示半径的字段 r 同名了。发生这种情况时，我们说，PlaneCircle 类的 r 字段*遮盖*了 Circle 类的 r 字段。（当然，这个例子是故意这么做的。新字段其实应该命名为 distanceFromOrigin。）

> 在你编写的代码中，为字段命名时应该避免遮盖超类的字段。如果遮盖了，几乎就表明代码写得不好。

这样定义 PlaneCircle 类之后，表达式 r 和 this.r 都引用 PlaneCircle 类中的这个字段。那么，如何引用 Circle 类中保存圆的半径的 r 字段呢？有一种特殊的句法可以实现这个需求——使用 super 关键字：

```
r       // 引用PlaneCircle的字段
this.r  // 引用PlaneCircle的字段
super.r // 引用Circle的字段
```

引用被遮盖的字段还有一种方式——把 this（或类的实例）校正为适当的超类，然后再访问字段：

```
((Circle) this).r // 引用Circle类的字段
```

如果想引用的遮盖字段不是在类的直接超类中定义的，这种校正技术特别有用。假如有三

个类 A、B 和 C，它们都定义了一个名为 x 的字段，而且 C 是 B 的子类，B 是 A 的子类。那么，在 C 类的方法中可以按照下面的方式引用这些不同的字段：

```
x               // C类的x字段
this.x          // C类的x字段
super.x         // B类的x字段
((B)this).x     // B类的x字段
((A)this).x     // A类的x字段
super.super.x   // 非法,不能这样引用A类的x字段
```

不能使用 super.super.x 引用超类的超类中的遮盖字段 x。这种句法不合法。

类似地，如果 c 是 C 类的实例，那么可以像这样引用这三个字段：

```
c.x            // C类的x字段
((B)c).x       // B类的x字段
((A)c).x       // A类的x字段
```

目前为止，讨论的都是实例字段。类字段也能被遮盖。引用被遮盖的类字段中的值，可以使用相同的 super 句法，但没必要这么做，因为始终可以把类名放在类字段前引用这个字段。假如 PlaneCircle 的实现方觉得 Circle.PI 字段没有提供足够的小数位，那么他可以自己定义 PI 字段：

```
public static final double PI = 3.14159265358979323846;
```

现在，PlaneCircle 类中的代码可以通过表达式 PI 或 PlaneCircle.PI 使用这个更精确的值，还可以使用表达式 super.PI 和 Circle.PI 引用精度不高的旧值。不过，PlaneCircle 继承的 area() 和 circumference() 方法是在 Circle 类中定义的，所以，就算 Circle.PI 被 PlaneCircle.PI 遮盖了，这两个方法还是会使用 Circle.PI 的值。

3.4.6 覆盖超类的方法

如果类中定义的某个实例方法和超类的某个方法有相同的名称、返回值类型和参数，那么这个方法会*覆盖*（override）超类中对应的方法。在这个类的对象上调用这个方法时，调用的是新定义的方法，而不是超类中定义的旧方法。

覆盖方法的返回值类型可以是原方法返回值的子类（没必要一模一样）。这叫作*协变返回*（covariant return）。

方法覆盖是面向对象编程中一项重要且有用的技术。PlaneCircle 没有覆盖 Circle 类定义的任何方法，不过，假设我们要再定义一个 Circle 的子类，名为 Ellipse。

此时，Ellipse 一定要覆盖 Circle 的 area() 和 circumference() 方法，因为计算圆的面积和周长的公式不适用于椭圆。

下面针对方法覆盖的讨论只涉及实例方法。类方法的运作机制完全不同，无法覆盖。和字段一样，类方法也能被子类遮盖，但不能覆盖。本章前面说过，好的编程风格是调用类方法时始终在前面加上定义这个方法的类名。如果把类名当成方法名的一部分，那么这两个方法的名称就不一样，因此其实并没有遮盖什么。

在进一步讨论方法覆盖之前，要理解方法覆盖和方法重载之间的区别。第 2 章说过，方法重载指的是（在同一个类中）定义多个名称相同但参数列表不同的方法。这和方法覆盖十分不同，因此别混淆了。

1. 覆盖不是遮盖

虽然 Java 使用很多类似的方式对待字段和方法，但方法覆盖和字段遮盖一点儿都不一样。为了引用遮盖的字段，只需把对象校正成适当超类的实例，但不能使用这种技术调用覆盖的实例方法。下述代码展示了这个重要区别：

```
class A {                          // 定义一个类,名为A
  int i = 1;                       // 一个实例字段
  int f() { return i; }            // 一个实例方法
  static char g() { return 'A'; }  // 一个类方法
}

class B extends A {                // 定义A的一个子类
  int i = 2;                       // 遮盖A类的字段i
  int f() { return -i; }           // 覆盖A类的方法f
  static char g() { return 'B'; }  // 遮盖A类的类方法g()
}

public class OverrideTest {
  public static void main(String args[]) {
    B b = new B();               // 创建一个类型为B的新对象
    System.out.println(b.i);     // 引用B.i,打印2
    System.out.println(b.f());   // 引用B.f(),打印-2
    System.out.println(b.g());   // 引用B.g(),打印B
    System.out.println(B.g());   // 调用B.g()更好的方式

    A a = (A) b;                 // 把b校正成A类的实例
    System.out.println(a.i);     // 现在引用的是A.i,打印1
    System.out.println(a.f());   // 还是引用B.f(),打印-2
    System.out.println(a.g());   // 引用A.g(),打印A
    System.out.println(A.g());   // 调用A.g()更好的方式
  }
}
```

初看起来，可能觉得方法覆盖和字段遮盖的这种区别有点奇怪，但稍微想想，确实有道理。

假设我们要处理一些 `Circle` 和 `Ellipse` 对象。为了记录这些圆和椭圆，我们把它们存储在一个 `Circle[]` 类型的数组中。这么做是可以的，因为 `Ellipse` 是 `Circle` 的子类，所以所有 `Ellipse` 对象都是合法的 `Circle` 对象。

遍历这个数组的元素时，不需要知道也无需关心元素是 `Circle` 对象还是 `Ellipse` 对象。不过，需要密切关注的是，在数组的元素上调用 `area()` 方法是否能得到正确的值。也就是说，如果是椭圆对象就不能使用计算圆面积的公式。

我们真正希望的是，计算面积时对象能“做正确的事”：`Circle` 对象使用自己的方式计算，`Ellipse` 对象使用对椭圆来说正确的方式计算。

这样理解，就不会对 Java 使用不同的方式处理方法覆盖和字段遮盖感到奇怪了。

2. 虚拟方法查找

如果一个 `Circle[]` 类型的数组保存的是 `Circle` 和 `Ellipse` 对象，那么编译器怎么知道要在具体的元素上调用 `Circle` 类还是 `Ellipse` 类的 `area()` 方法呢？事实上，源码编译器在编译时并不知道要调用哪个方法。

不过，`javac` 生成的字节码会在运行时使用“虚拟方法查找”（virtual method lookup）。解释器运行代码时，会查找适用于数组中各个对象的 `area()` 方法。即，解释器解释表达式 `o.area()` 时，会检查变量 `o` 引用的对象的真正运行时类型，然后找到适用于这个类型的 `area()` 方法。

> 某些其他语言（例如 C# 和 C++）默认不使用虚拟查找，如果程序员想在子类中覆盖方法，要显式使用 `virtual` 关键字。

JVM 不会直接使用关联在变量 `o` 表示的静态类型身上的 `area()` 方法，如果这么做，前面详述的方法覆盖机制就不成立了。Java 的实例方法默认使用虚拟查找。第 4 章会详细介绍编译时和运行时类型，以及它们对虚拟方法查找的影响。

3. 调用被覆盖的方法

我们已经说明了方法覆盖和字段遮盖之间的重要区别。然而，调用被覆盖的方法的 Java 句法和访问被遮盖的字段的句法十分类似——都使用 `super` 关键字。如下述代码所示：

```
class A {
  int i = 1;              // 被子类B遮盖的实例字段
  int f() { return i; }   // 被子类B覆盖的实例方法
```

```
}

class B extends A {
  int i;                    // 这个字段遮盖A类的字段i
  int f() {                 // 这个方法覆盖A类的方法f()
    i = super.i + 1;      // 可以像这样读取A.i的值
    return super.f() + i; // 可以像这样调用A.f()
  }
}
```

前面说过，使用 super 引用被遮盖的字段时，相当于把 this 校正为超类类型，然后通过超类类型访问字段。不过，使用 super 调用被覆盖的方法和校正 this 引用不是一回事。也就是说，在上述代码中，表达式 super.f() 和 ((A)this).f() 的作用不一样。

解释器使用 super 句法调用实例方法时，会执行一种修改过的虚拟方法查找。第一步和常规的虚拟方法查找一样，确定调用方法的对象属于哪个类。正常情况下，运行时会在这个类中寻找对应的方法定义。但是，使用 super 句法调用方法时，先在这个类的超类中查找。如果超类直接实现了这个方法，那就调用这个方法。如果超类继承了这个方法，那就调用继承的方法。

注意，super 关键字调用的是方法的直接覆盖版本。假设 A 类有个子类 B，B 类有个子类 C，而且这三个类都定义了同一个方法 f()。在 C.f() 方法中使用 super.f() 可以调用方法 B.f()，因为 C.f() 直接覆盖了 B.f()。但是，C.f() 不能直接调用 A.f()，因为 super.super.f() 不是合法的 Java 句法。当然，如果 C.f() 调用了 B.f()，有合理的理由认为，B.f() 可能会调用 A.f()。

使用被覆盖的方法时，这种链式调用相当常见。覆盖方法是增强方法功能，但不完全取代这个方法的一种方式。

> 别把调用被覆盖方法的 super 和构造方法中调用超类构造方法的 super() 搞混了。虽然二者使用的关键字相同，但却是两种完全不同的句法。具体而言，可以在类中的任何位置使用 super 调用超类中被覆盖的方法，但是只能在构造方法的第一个语句中使用 super() 调用超类的构造方法。

还有一点很重要，即记住，只能在覆盖某个方法的类内部使用 super 调用被覆盖的方法。假如 e 引用的是一个 Ellipse 对象，那么无法在 e 上调用 Circle 类中定义的 area() 方法。

3.5 数据隐藏和封装

本章开头说过，类由一些数据和方法组成。目前，我们尚未说明的最重要的面向对象技术之一是，把数据隐藏在类中，只能通过方法获取。这种技术叫作封装（encapsulation），因为它把数据（和内部方法）安全地密封在类这个“容器”中，只能由可信的用户（即这个

类中的方法）访问。

为什么要这么做呢？最重要的原因是，隐藏类的内部实现细节。如果避免让程序员依赖这些细节，你就可以放心地修改实现，而无需担心会破坏使用这个类的现有代码。

> 你应该始终封装自己的代码。如果没有封装好，那么几乎无法推知并最终确认代码是否正确，尤其是在多线程环境中（而基本上所有 Java 程序都运行在多线程环境中）。

使用封装的另一个原因是保护类，避免有意或无意做了糊涂事。类中经常包含一些相互依赖的字段，而且这些字段的状态必须始终如一。如果允许程序员（包括你自己）直接操作这些字段，修改某个字段后可能不会修改重要的相关字段，那么类的状态就前后不一致了。然而，如果必须调用方法才能修改字段，那么这个方法可以做一切所需的措施，确保状态一致。类似地，如果类中定义的某些方法仅供内部使用，隐藏这些方法能避免这个类的用户调用这些方法。

封装还可以这样理解：把类的数据都隐藏后，方法就是在这个类的对象上能执行的唯一一种可能的操作。

只要小心测试和调试方法，就可以认为类能按预期的方式运行。然而，如果类的所有字段都可以直接操作，那么要测试的可能性根本数不完。

> 这种想法可以得到一个非常重要的推论，5.5 节介绍 Java 程序的安全性时会说明（Java 程序的安全和 Java 编程语言的类型安全不是同一个概念）。

隐藏类的字段和方法还有一些次要的原因。

- 如果内部字段和方法在外部可见，会弄乱类的 API。让可见的字段尽量少，可以保持类的整洁，从而更易于使用和理解。
- 如果方法对类的使用者可见，就必须为其编写文档。把方法隐藏起来，可以节省时间和精力。

3.5.1 访问控制

Java 定义了一些访问控制规则，可以禁止类的成员在类外部使用。在本章的一些示例中，你已经见过字段和方法声明中使用的 `public` 修饰符。这个 `public` 关键字，连同 `protected` 和 `private`（还有一个特殊的），是访问控制修饰符，为字段或方法指定访问规则。

1. 访问包

Java 语言不直接支持包的访问控制。访问控制一般在类和类的成员这些层级完成。

> 已经加载的包始终可以被同一个包中的代码访问。一个包在其他包中是否能访问，取决于这个包在宿主系统中的部署方式。例如，如果组成包的类文件存储在一个目录中，那么用户必须能访问这个目录和其中的文件才能访问包。

2. 访问类

默认情况下，顶层类在定义它的包中可以访问。不过，如果顶层类声明为 `public`，那么在任何地方都能访问。

> 第 4 章会介绍嵌套类。嵌套类是定义为其他类的成员的类。因为这种内部类是某个类的成员，因此也遵守成员的访问控制规则。

3. 访问成员

类的成员在类的主体里始终可以访问。默认情况下，在定义这个类的包中也可以访问成员。这种默认的访问等级一般叫作*包访问*。这只是四个可用的访问等级中的一个。其他三个等级使用 `public`、`protected` 和 `private` 修饰符定义。下面是使用这三个修饰符的示例代码：

```
public class Laundromat {       // 所有人都可以使用这个类
  private Laundry[] dirty;      // 不能使用这个内部字段
  public void wash() { ... }    // 但能使用这两个公开的方法
  public void dry() { ... }     // 处理内部字段
  // 子类可能会想调整这个字段
  protected int temperature;
}
```

下述访问规则适用于类的成员。

- 类中的所有字段和方法在类的主体里始终可以使用。
- 如果类的成员使用 `public` 修饰符声明，那么可以在能访问这个类的任何地方访问这个成员。这是限制最松的访问控制类型。
- 如果类的成员声明为 `private`，那么除了在类内部之外，其他地方都不能访问这个成员。这是限制最严的访问控制类型。
- 如果类的成员声明为 `protected`，那么包里的所有类都能访问这个成员（等同于默认的包访问规则），而且在这个类的任何子类的主体中也能访问这个成员，而不管子类在哪个包中定义。

- 如果声明类的成员时没使用任何修饰符，那么使用默认的访问规则（有时叫包访问），包中的所有类都能访问这个成员，但在包外部不能访问。

> 默认的访问规则比 protected 严格，因为默认规则不允许在包外部的子类中访问成员。

使用 protected 修饰的成员时要格外小心。假设 A 类使用 protected 声明了一个字段 x，而且在另一个包中定义的 B 类继承 A 类（重点是 B 类在另一包中定义）。因此，B 类继承了这个 protected 声明的字段 x，那么，在 B 类的代码中可以访问当前实例的这个字段，而且引用 B 类实例的代码也能访问这个字段。但是，这并不意味着在 B 类的代码中能读取任何一个 A 类实例的受保护字段。

下面通过代码讲解这个语言细节。A 类的定义如下：

```
package javanut6.ch03;

public class A {
    protected final String name;

    public A(String named) {
        name = named;
    }

    public String getName() {
        return name;
    }
}
```

B 类的定义如下：

```
package javanut6.ch03.different;

import javanut6.ch03.A;

public class B extends A {

    public B(String named) {
        super(named);
    }

    @Override
    public String getName() {
        return "B: " + name;
    }
}
```

> Java 的包不能“嵌套”，所以 javanut6.ch03.different 和 javanut6.ch03 是不同的包。javanut6.ch03.different 不以任何方式包含在 javanut6.ch03 中，也和 javanut6.ch03 没有任何关系。

可是，如果我们试图把下面这个新方法添加到 B 类中，会导致编译出错，因为 B 类的实例无法访问任何一个 A 类的实例：

```
public String examine(A a) {
    return "B sees: " + a.name;
}
```

如果把这个方法改成：

```
public String examine(B b) {
    return "B sees another B: " + b.name;
}
```

就能编译通过，因为同一类型的多个实例可以访问各自的 protected 字段。当然，如果 B 类和 A 类在同一包中，那么任何一个 B 类的实例都能访问任何一个 A 类实例的全部受保护字段，因为使用 protected 声明的字段对同一个包中的每个类都可见。

4. 访问控制和继承

Java 规范规定：

- 子类继承超类中所有可以访问的实例字段和实例方法；
- 如果子类和超类在同一个包中定义，那么子类继承所有没使用 private 声明的实例字段和方法；
- 如果子类在其他包中定义，那么它继承所有使用 protected 和 public 声明的实例字段和方法；
- 使用 private 声明的字段和方法绝不会被继承；类字段和类方法也一样；
- 构造方法不会被继承（而是链在一起调用，本章前面已经说过）。

不过，有些程序员会对“子类不继承超类中不可访问的字段和方法”感到困惑。这似乎暗示了，创建子类的实例时不会为超类中使用 private 声明的字段分配内存。然而，这不是上述规定想表述的。

> 其实，子类的每个实例都包含一个完整的超类实例，其中包括所有不可访问的字段和方法。

某些成员可能无法访问，这似乎和类的成员在类的主体中始终可以访问相矛盾。为了避免

误解，我们要使用“继承的成员”表示那些可以访问的超类成员。

那么，关于成员访问性的正确表述应该是：“所有继承的成员和所有在类中定义的成员都是可以访问的。”这句话还可以换种方式说：

- 类继承超类的所有实例字段和实例方法（但不继承构造方法）；
- 在类的主体中始终可以访问这个类定义的所有字段和方法，而且还可以访问继承自超类的可访问的字段和方法。

5. 成员访问规则总结

表 3-1 总结了成员的访问规则。

表3-1：类中成员的可访问性

能否访问	成员可见性			
	公开	受保护	默认	私有
定义成员的类	是	是	是	是
同一个包中的类	是	是	是	否
不同包中的子类	是	是	否	否
不同的包，也不是子类	是	否	否	否

下面是一些使用可见性修饰符的经验法则。

- 只使用 `public` 声明组成类的公开 API 的方法和常量。使用 `public` 声明的字段只能是常量和不能修改的对象，而且必须同时使用 `final` 声明。
- 使用 `protected` 声明大多数使用这个类的程序员不会用到的字段和方法，但在其他包中定义子类时可能会用到。

> 严格来说，使用 `protected` 声明的成员是类公开 API 的一部分，必须为其编写文档，而且不能轻易修改，以防破坏依赖这些成员的代码。

- 如果字段和方法供类的内部实现细节使用，但是同一个包中协作的类也要使用，那么就使用默认的包可见性。
- 使用 `private` 声明只在类内部使用，在其他地方都要隐藏的字段和方法。

如果不确定该使用 `protected`、包还是 `private` 可见性，那么先使用 `private`。如果太过严格，可以稍微放松访问限制（如果是字段的话，还可以提供访问器方法）。

设计 API 时这么做尤其重要，因为提高访问限制是不向后兼容的改动，可能会破坏依赖成员访问性的代码。

3.5.2 数据访问器方法

在 Circle 类那个示例中，我们使用 public 声明表示圆半径的字段。Circle 类可能有很好的理由让这个字段可以公开访问；这个类很简单，字段之间不相互依赖。但是，当前实现的 Circle 类允许对象的半径为负数，而半径为负数的圆肯定不存在。可是，只要半径存储在声明为 public 的字段中，任何程序员都能把这个字段的值设为任何想要的值，而不管这个值有多么不合理。唯一的办法是限制程序员，不让他们直接访问这个字段，然后定义 public 方法，间接访问这个字段。提供 public 方法读写字段和把字段本身声明为 public 不是一回事。目前而言，二者的区别是，方法可以检查错误。

例如，我们或许不想让 Circle 对象的半径使用负数——负数显然不合理，但目前的实现没有阻止这么做。示例 3-4 展示了如何修改 Circle 类的定义，避免把半径设为负数。

Circle 类的这个版本使用 protected 声明 r 字段，还定义了访问器方法 getRadius() 和 setRadius()，用于读写这个字段的值，而且限制半径不能为负数。r 字段使用 protected 声明，所以可以在子类中直接（且高效地）访问。

示例 3-4：使用数据隐藏和封装技术定义的 Circle 类

```
package shapes;        // 为这个类指定一个包

public class Circle {  // 这个类还使用public声明
  // 这是通用的常量,所以要保证声明为public
  public static final double PI = 3.14159;

  protected double r;  // 半径被隐藏了,但在子类中可见

  // 限制半径取值的方法
  // 这是子类可能感兴趣的实现细节
  protected void checkRadius(double radius) {
    if (radius < 0.0)
      throw new IllegalArgumentException("radius may not be negative.");
  }

  // 非默认的构造方法
  public Circle(double r) {
    checkRadius(r);
    this.r = r;
  }

  // 公开的数据访问器方法
  public double getRadius() { return r; }
  public void setRadius(double r) {
    checkRadius(r);
    this.r = r;
  }

  // 操作实例字段的方法
  public double area() { return PI * r * r; }
  public double circumference() { return 2 * PI * r; }
}
```

我们在一个名为 shapes 的包中定义 Circle 类。因为 r 字段使用 protected 声明，所以 shapes 包中的任何其他类都能直接访问这个字段，而且能把它设为任何值。这里假设 shapes 包中的所有类都由同一个作者或者协作的多个作者编写，而且包中的类相互信任，不会滥用拥有的访问权限影响彼此的实现细节。

最后，限制半径不能使用负数的代码在一个使用 protected 声明的方法中，这个方法是 checkRadius()。虽然 Circle 类的用户无法调用这个方法，但这个类的子类可以调用，而且如果想修改对半径的限制，还可以覆盖这个方法。

> 在 Java 中，数据访问器方法的命名有个通用约定，即以“get”和“set”开头。但是，如果要访问的字段是 boolean 类型，那么读取字段的方法使用的名称可能会以“is”开头。例如，名为 readable 的 boolean 类型字段对应的访问器方法是 isReadable() 而不是 getReadable()。

3.6　抽象类和方法

在示例 3-4 中，我们把 Circle 类声明为 shapes 包的一部分。假设我们计划实现多个表示形状的类：Rectangle、Square、Ellipse、Triangle 等。我们可以在这些表示形状的类中定义两个基本方法：area() 和 circumference()。那么，为了能方便处理由形状组成的数组，这些表示形状的类最好有个共同的超类 Shape。这样组织类层次结构的话，每个形状对象，不管具体表示的是什么形状，都能赋予类型为 Shape 的变量、字段或数组元素。我们想在 Shape 类中封装所有形状共用的功能（例如，area() 和 circumference() 方法）。但是，通用的 Shape 类不表示任何类型的形状，所以不能为这些方法定义有用的实现。Java 使用*抽象方法*解决这种问题。

Java 允许使用 abstract 修饰符声明方法，此时只定义方法但不实现方法。abstract 修饰的方法没有主体，只有一个签名和一个分号。[2] 以下是 abstract 方法和这些方法所在的 abstract 类相关的规则。

- 只要类中有一个 abstract 方法，那么这个类本身就自动成为 abstract 类，而且必须声明为 abstract 类，否则会导致编译出错。
- abstract 类无法实例化。
- abstract 类的子类必须覆盖超类的每个 abstract 方法并且把这些方法全部实现（即提供方法主体），才能实例化。这种类一般叫作*具体子类*（concrete subclass），目的是强调它不是抽象类。

注 2：Java 中的抽象方法和 C++ 中的纯虚拟函数（即声明为 = 0 的虚拟函数）有点像。在 C++ 中，包含纯虚拟函数的类是抽象类，不能实例化。包含抽象方法的 Java 类也一样不能实例化。

- 如果 abstract 类的子类没有实现继承的所有 abstract 方法，那么这个子类还是抽象类，而且必须使用 abstract 声明。
- 使用 static、private 和 final 声明的方法不能是抽象方法，因为这三种方法在子类中不能覆盖。类似地，final 类中不能有任何 abstract 方法。
- 就算类中没有 abstract 方法，这个类也能声明为 abstract。使用这种方式声明的 abstract 类表明实现的不完整，要交给子类实现。这种类不能实例化。

> 第 11 章会见到 Classloader 类，这个类就没有任何抽象方法。

下面通过一个示例说明这些规则的运作方式。如果定义 Shape 类时把 area() 和 circumference() 声明为 abstract 方法，那么 Shape 的子类必须实现这两个方法才能实例化。也就是说，每个 Shape 对象都要确保实现了这两个方法。示例 3-5 展示了如何编写代码。在这段代码中，定义了一个抽象的 Shape 类和两个具体子类。

示例 3-5：一个抽象类和两个具体子类

```
public abstract class Shape {
  public abstract double area();                      // 两个抽象方法
  public abstract double circumference();             // 注意，没有主体，只有分号
}

class Circle extends Shape {
  public static final double PI = 3.14159265358979323846;
  protected double r;                                 // 实例字段
  public Circle(double r) { this.r = r; }             // 构造方法
  public double getRadius() { return r; }             // 访问器
  public double area() { return PI*r*r; }             // 实现超类中的
  public double circumference() { return 2*PI*r; }    // 两个抽象方法
}

class Rectangle extends Shape {
  protected double w, h;                              // 实例字段
  public Rectangle(double w, double h) {              // 构造方法
    this.w = w; this.h = h;
  }
  public double getWidth() { return w; }              // 访问器方法
  public double getHeight() { return h; }             // 另一个访问器
  public double area() { return w*h; }                // 实现超类中的
  public double circumference() { return 2*(w + h); } // 两个抽象方法
}
```

Shape 类中每个抽象方法的括号后面都是分号，没有花括号，也没定义方法的主体。使用示例 3-5 中定义的这几个类可以编写如下的代码：

```
Shape[] shapes = new Shape[3];        // 创建一个保存形状的数组
shapes[0] = new Circle(2.0);          // 填充这个数组
shapes[1] = new Rectangle(1.0, 3.0);
shapes[2] = new Rectangle(4.0, 2.0);

double totalArea = 0;
for(int i = 0; i < shapes.length; i++)
    totalArea += shapes[i].area();    // 计算这些形状的面积
```

有两点要注意。

- Shape 类的子类对象可以赋值给 Shape 类型数组中的元素，无需校正。这又是一个放大转换引用类型（第 2 章讨论过）的例子。
- 即便 Shape 类没有定义 area() 和 circumference() 方法的主体，各个 Shape 对象还是能调用这两个方法。调用这两个方法时，使用虚拟方法查找技术找到要调用的方法。因此，圆的面积使用 Circle 类中定义的方法计算，矩形的面积使用 Rectangle 类中定义的方法计算。

转换引用类型

对象可以在不同的引用类型之间转换。和基本类型一样，引用类型转换可以是放大转换（编译器自动完成），也可以是需要校正的缩小转换（或许运行时还要检查）。要想理解引用类型的转换，必须理解引用类型组成的层次结构，这个体系叫作*类层次结构*。

每个 Java 引用类型都*扩展*其他类型，被扩展的类型是这个类型的*超类*。类型继承超类的字段和方法，然后定义属于自己的一些额外的字段和方法。在 Java 中，类层次结构的根是一个特殊的类，名为 Object。所有 Java 类都直接或间接地扩展 Object 类。Object 类定义了一些特殊的方法，所有对象都能继承（或覆盖）这些方法。

预定义的 String 类和本章前面定义的 Point 类都扩展 Object 类。因此，可以说，所有 String 对象也都是 Object 对象。也可以说，所有 Point 对象都是 Object 对象。但是，反过来说就不对了。我们不能说每个 Object 对象都是 String 对象，因为如前所示，有些 Object 对象是 Point 对象。

简单理解类层次结构之后，我们可以定义引用类型的转换规则了。

- 对象不能转换成不相关的类型。例如，就算使用校正运算符，Java 编译器也不允许把 String 对象转换成 Point 对象。
- 对象可以转换成超类类型，或者任何祖先类类型。这是放大转换，因此不用校正。例如，String 对象可以赋值给 Object 类型的变量，或者传入期待 Object 类型参数的方法。

其实没有执行转换操作，而是直接把对象当成超类的实例。这种行为有时称为里氏替换原则（Liskov substitution principle），以第一个明确表述这种行为的计算机科学家 Barbara Liskov 的名字命名。

- 对象可以转换成子类类型，但这是缩小转换，需要校正。Java 编译器临时允许执行这种转换，但 Java 解释器在运行时会做检查，确保转换有效。根据程序的逻辑，确认对象的确是子类的实例后才会把对象校正成子类类型。否则，解释器会抛出 `ClassCastException` 异常。例如，如果把一个 `String` 对象赋值给 `Object` 类型的变量，那么后面可以校正这个变量的值，再变回 `String` 类型：

```
Object o = "string";   // 把String对象放大转换成Object类型
String s = (String) o; // 程序后面再把这个Object对象缩小转换成String类型
```

数组是对象，而且有自己的一套转换规则。首先，任何数组都能放大转换成 `Object` 对象。带校正的缩小转换能把这个对象转换回数组。下面是一个示例：

```
// 把数组放大转换成Object对象
Object o = new int[] {1,2,3};
// 程序后面……

int[] a = (int[]) o;  // 缩小转换回数组类型
```

除了能把数组转换成对象之外，如果两个数组的“基类型”是可以相互转换的引用类型，那么数组还能转换成另一个类型的数组。例如：

```
// 这是一个字符串数组
String[] strings = new String[] { "hi", "there" };
// 可以放大转换成CharSequence[]类型
// 因为String类型可以放大转换成CharSequence类型
CharSequence[] sequences = strings;
// 缩小转换回String[]类型需要校正
strings = (String[]) sequences;
// 这是一个由字符串数组组成的数组
String[][] s = new String[][] { strings };
// 不能转换成CharSequence[]类型,因为String[]类型
// 不能转换成CharSequence类型:维数不匹配

sequences = s; // 不会编译这行代码
// s可以转换成Object类型或Object[]类型,因为所有数组类型
// (包括String[]和String[][]类型)都能转换成Object类型
Object[] objects = s;
```

注意，这些数组转换规则只适用于由对象或数组组成的数组。基本类型的数组不能转换为任何其他数组类型，就算基本基类型之间能相互转换也不行：

```
// 就算int类型能放大转换成double类型
// 也不能把int[]类型转换成double[]类型
```

```
// 这行代码会导致编译出错
double[] data = new int[] {1,2,3};
// 但是,这行代码是合法的,因为int[]类型能转换成Object类型
Object[] objects = new int[][] {{1,2},{3,4}};
```

3.7 修饰符总结

如前所示，类、接口和它们的成员都能使用一个或多个修饰符声明——这些修饰符是 `public`、`static` 和 `final` 等关键字。下面对本章做个总结，列出所有 Java 修饰符，说明各自能修饰的 Java 结构种类和作用。详情如表 3-2 所示。还可以参阅 3.1 节、3.2.1 节和 2.6.2 节。

表3-2：Java修饰符

修饰符	用　于	意　义
`abstract`	类	这个类不能实例化，而且可能包含未实现的方法
	接口	所有接口都是抽象的。声明接口时这个修饰符是可选的
	方法	这个方法没有主体，主体由子类提供。签名后面是一个分号。所在的类必须也是抽象的
`default`	方法	这个接口方法的实现是可选的。接口为不想实现这个方法的类提供了一个默认实现。详情参见第 4 章
`final`	类	不能创建这个类的子类
	方法	不能覆盖这个方法
	字段	这个字段的值不能改变。`static final` 修饰的字段是编译时常量
	变量	值不能改变的局部变量、方法参数或异常参数
`native`	方法	这个方法使用某种与平台无关的方式实现（经常使用 C 语言）。没有提供主体，签名后面是一个分号
无（包）	类	没声明为 `public` 的类只能在包中访问
	接口	没声明为 `public` 的接口只能在包中访问
	成员	没声明为 `private`、`protected` 或 `public` 的成员具有包可见性，只能在包中访问
`private`	成员	这个成员只在定义它的类中可以访问
`protected`	成员	这个成员只在定义它的包中和子类中可以访问
`public`	类	能访问所在包的地方都能访问这个类
	接口	能访问所在包的地方都能访问这个接口
	成员	能访问所在类的地方都能访问这个成员
`strictfp`	类	这个类中的所有方法都隐式声明为 `strictfp`
	方法	这个方法必须使用严格遵守 IEEE 754 标准的方式执行浮点运算。具体而言，所有数值，包括中间结果，都要使用 IEEE`float` 或 `double` 类型表示，而且不能利用本地平台浮点格式或硬件提供的额外精度或取值范围。这个修饰符极少使用
`static`	类	使用 `static` 声明的内部类是顶层类，而不是所在类的成员。详情参见第 4 章
	方法	`static` 方法是类方法。不隐式传入 `this` 对象引用。可通过类名调用

（续）

修饰符	用　　于	意　　义
	字段	static 字段是类字段。不管创建多少类实例，这个字段都只有一个实例。可通过类名访问
	初始化程序	这个初始化程序在加载类时运行，而不是创建实例时运行
synchronized	方法	这个方法对类或实例执行非原子操作，所以必须小心，确保不能让两个线程同时修改类或实例。对 static 方法来说，执行方法之前先为类获取一个锁。对非 static 方法来说，会为具体的对象实例获取一个锁。详情参见第 5 章
transient	字段	这个字段不是对象持久化状态的一部分，因此不会随对象一起序列化。在对象序列化时使用，参见 java.io.ObjectOutputStream
volatile	字段	这个字段能被异步线程访问，因此必须对其做些特定的优化。这个修饰符有时可以替代 synchronized。详情参见第 5 章

第 4 章

Java类型系统

本章以基于类的面向对象编程为基础，介绍高效使用 Java 静态类型系统所需知道的其他概念。

> *静态类型*语言的变量类型是确定的，如果把不兼容类型的值赋给变量，会导致编译时错误。Java 是一种静态类型语言。只在运行时检查类型兼容性的语言叫作*动态类型语言*，JavaScript 便是一种动态类型语言。

Java 的类型系统不仅涉及类和基本类型，还涉及与类的基本概念相关的其他引用类型，但这些引用类型有些不同，`javac` 或 JVM 往往会使用特殊的方式处理。

我们已经介绍了数组和类，它们是使用最广泛的两种 Java 引用类型。本章先介绍另一种重要的引用类型——接口。然后介绍 Java 的*泛型*，泛型在 Java 的类型系统中扮演着重要角色。掌握这些知识后，我们再介绍 Java 中编译时和运行时类型之间的区别。

为了完整介绍 Java 的引用类型，我们要介绍两种特殊的类和接口——*枚举*和*注解*。本章最后介绍*嵌套类型*和 Java 8 引入的 lambda *表达式*。

下面先介绍接口。接口算是继类之后最重要的 Java 引用类型，而且是整个 Java 类型系统的重要组成。

4.1 接口

第 3 章介绍了继承这个概念。我们知道，一个 Java 类只能继承一个类。这对我们要编写的面向对象程序来说是个相当严格的限制。Java 的设计者知道这一点，但他们也是为了确保 Java 实现面向对象编程的方式比其他语言（例如 C++）简单。

他们选择的方式是提出接口这个概念。和类一样，*接口*定义一种新的引用类型。如“接口”这个名称所示，接口的作用只是描绘 API，因此，接口提供类型的描述信息，以及实现这个 API 的类应该提供的方法（和签名）。

一般来说，Java 的接口不为它描述的方法提供实现代码。这些方法是*强制*要实现的——想实现接口的类必须实现这些方法。

不过，接口可能想把 API 中的某些方法标记为可选，如果实现接口的类不想实现就不用实现。这种机制通过 `default` 关键字实现，接口必须为可选的方法提供*默认实现*，未实现这些方法的类会使用默认实现。

> 接口中的可选方法是 Java 8 的新功能，之前的版本中没有。4.1.5 节会完整介绍如何使用可选方法（也叫默认方法）。

接口不能直接实例化，也不能创建这种接口类型的成员。接口必须通过类实现，而且类要提供所需的方法主体。

这个类的实例既属于这个类定义的类型，也属于这个接口定义的类型。不属于同一个类或超类的对象，通过实现同一个接口，也能属于同一种类型。

4.1.1 定义接口

定义接口的方式和定义类差不多，不过所有（非默认的）方法都是抽象方法，而且关键字 `class` 要换成 `interface`。例如，下述代码定义了一个名为 `Centered` 的接口。第 3 章定义的 `Shape` 类如果想设定和读取形状的中心点坐标，就可以实现这个接口。

```
interface Centered {
  void setCenter(double x, double y);
  double getCenterX();
  double getCenterY();
}
```

接口的成员有些限制。

- 接口中所有强制方法都隐式使用 `abstract` 声明，不能有方法主体，要使用分号。可以使用 `abstract` 修饰符，但一般习惯省略。
- 接口定义公开的 API。接口中的所有成员都隐式使用 `public` 声明，而且习惯省略不必要的 `public` 修饰符。如果在接口中使用 `protected` 或 `private` 定义方法，会导致编译时错误。
- 接口不能定义任何实例字段。字段是实现细节，而接口是规格不是实现。在接口中只能定义同时使用 `static` 和 `final` 声明的常量。
- 接口不能实例化，因此不定义构造方法。
- 接口中可以包含嵌套类型。嵌套类型隐式使用 `public` 和 `static` 声明。4.4 节会完整介绍嵌套类型。
- 从 Java 8 开始，接口中可以包含静态方法。Java 之前的版本不允许这么做，这被广泛认为是 Java 语言的一个设计缺陷。

4.1.2　扩展接口

接口可以扩展其他接口，而且和类的定义一样，接口的定义可以包含一个 `extends` 子句。接口扩展另一个接口时，会继承父接口中的所有方法和常量，而且可以定义新方法和常量。不过，和类不同的是，接口的 `extends` 子句可以包含多个父接口。例如，下述接口扩展了其他接口：

```
interface Positionable extends Centered {
  void setUpperRightCorner(double x, double y);
  double getUpperRightX();
  double getUpperRightY();
}
interface Transformable extends Scalable, Translatable, Rotatable {}
interface SuperShape extends Positionable, Transformable {}
```

扩展多个接口的接口，会继承每个父接口中的所有方法和常量，而且可以定义属于自己的方法和常量。实现这个接口的类必须实现这个接口直接定义的抽象方法，以及从所有父接口中继承的全部抽象方法。

4.1.3　实现接口

类使用 `extends` 指定超类，类似地，类使用 `implements` 列出它支持的一个或多个接口。`implements` 是一个 Java 关键字，可以出现在类声明中，但要放在 `extends` 子句后面。`implements` 关键字后面是这个类要实现的一组接口，接口之间使用逗号分隔。

类在 `implements` 子句中声明接口时，表明这个类要为接口中的每个强制方法提供实现（即主体）。如果实现接口的类没有为接口中的每个强制方法提供实现，那么这个类从接口中继承未实现的抽象方法，而且这个类本身必须使用 `abstract` 声明。如果类实现多个接口，

必须实现每个接口中的所有强制方法（否则这个类要使用 abstract 声明）。

下述代码展示了如何定义 CenteredRectangle 类，这个类扩展第 3 章定义的 Rectangle 类，而且实现 Centered 接口：

```
public class CenteredRectangle extends Rectangle implements Centered {
  // 新实例字段
  private double cx, cy;

  // 构造方法
  public CenteredRectangle(double cx, double cy, double w, double h) {
    super(w, h);
    this.cx = cx;
    this.cy = cy;
  }

  // 继承了Rectangle类中的所有方法
  // 但要为Centered接口中的所有方法提供实现
  public void setCenter(double x, double y) { cx = x; cy = y; }
  public double getCenterX() { return cx; }
  public double getCenterY() { return cy; }
}
```

假设我们按照 CenteredRectangle 类的实现方式实现了 CenteredCircle 和 CenteredSquare 类。每个类都扩展 Shape 类，所以如前所示，这些类的实例都可以当成 Shape 类的实例。因为每个类都实现了 Centered 接口，所以这些实例还可以当成 Centered 类型的实例。下述代码演示了对象既可以作为类类型的成员，也可以作为接口类型的成员：

```
Shape[] shapes = new Shape[3];     // 创建一个数组，保存形状对象

// 创建一些Centered类型的形状，存储在这个Shape[]类型的数组中
// 不用校正，因为都是放大转换
shapes[0] = new CenteredCircle(1.0, 1.0, 1.0);
shapes[1] = new CenteredSquare(2.5, 2, 3);
shapes[2] = new CenteredRectangle(2.3, 4.5, 3, 4);

// 计算这些形状的平均面积
// 以及到原点的平均距离
double totalArea = 0;
double totalDistance = 0;
for(int i = 0; i < shapes.length; i++) {
  totalArea += shapes[i].area();   // 计算这些形状的面积

  // 注意，一般来说，使用instanceof判断对象的运行时类型经常表明设计有问题
  if (shapes[i] instanceof Centered) { // 形状属于Centered类型
    // 注意，把Shape类型转换成Centered类型要校正
    //（不过，把CenteredSquare类型转换成Centered类型不用校正）
    Centered c = (Centered) shapes[i];

    double cx = c.getCenterX();    // 获取中心点的坐标
    double cy = c.getCenterY();    // 计算到原点的距离
    totalDistance += Math.sqrt(cx*cx + cy*cy);
```

```
    }
  }
  System.out.println("Average area: " + totalArea/shapes.length);
  System.out.println("Average distance: " + totalDistance/shapes.length);
```

在 Java 中，接口和类一样，也是数据类型。如果一个类实现了一个接口，那么这个类的实例可以赋值给这个接口类型的变量。

看过这个示例之后，别错误地认为必须先把 `CenteredRectangle` 对象赋值给 `Centered` 类型的变量才能调用 `setCenter()` 方法，或者要先赋值给 `Shape` 类型的变量才能调用 `area()` 方法。`CenteredRectangle` 类定义了 `setCenter()` 方法，而且从超类 `Rectangle` 中继承了 `area()` 方法，所以始终可以调用这两个方法。

4.1.4　实现多个接口

假设我们不仅想通过中心点摆放形状对象，也想通过右上角摆放形状对象，而且还想放大和缩小形状。还记得吗？虽然一个类只能扩展一个超类，但可以实现任意多个接口。假设我们已经定义好了合适的 `UpperRightCornered` 和 `Scalable` 接口，那么可以按照下述方式声明类：

```
public class SuperDuperSquare extends Shape
  implements Centered, UpperRightCornered, Scalable {
  // 类的成员省略了
}
```

一个类实现多个接口只是表明这个类要实现所有接口中的全部抽象方法（即强制方法）。

4.1.5　默认方法

Java 8 出现后，接口中的方法可以包含实现了。本节介绍这种方法——在接口描述的 API 中通过可选的方法表示，一般叫作*默认方法*。首先说明为什么需要这种默认机制。

1. 向后兼容性

Java 平台始终关注向后兼容性。这意味着，为前一版平台编写（或者已经编译）的代码在最新版平台中必须能继续使用。这个原则让开发团队坚信，升级 JDK 或 JRE 后不会破坏之前能正常运行的应用。

向后兼容性是 Java 平台的一大优势，但是为此，Java 平台有诸多约束。其中一个约束是，新发布的接口不能添加新的强制方法。

例如，假设我们要升级 `Positionable` 接口，添加获取和设定左下角顶点的功能：

```java
public interface Positionable extends Centered {
  void setUpperRightCorner(double x, double y);
  double getUpperRightX();
  double getUpperRightY();
  void setLowerLeftCorner(double x, double y);
  double getLowerLeftX();
  double getLowerLeftY();
}
```

重新定义接口之后，如果尝试在为旧接口编写的代码中使用这个新接口，不会成功，因为现有的代码中没有 `setLowerLeftCorner()`、`getLowerLeftX()` 和 `getLowerLeftY()` 这三个强制方法。

> 在你的代码中可以轻易地看到效果。编译一个依赖接口的类文件，在接口中添加一个新的强制方法，然后使用新版接口和旧的类文件尝试运行程序。你会看到程序崩溃，抛出 `NoClassDefError` 异常。

Java 8 的设计者注意到了这个缺陷，因为设计者的目标之一是升级 Java 核心中的集合库，引入使用 lambda 表达式的方法。

若想解决这个问题，需要一种新机制。这种机制必须要允许向接口中添加可选的新方法，而不破坏向后兼容性。

2. 实现默认方法

在接口中添加新方法而不破坏向后兼容性，这需要为接口的旧实现提供一些新实现，以便接口能继续使用。这个机制是默认方法，在 JDK 8 中首次添加到 Java 平台。

> 默认方法（有时也叫可选方法）可以添加到任何接口中。默认方法必须包含实现，即默认实现，写在接口定义中。

默认方法的基本行为如下：

- 实现接口的类可以（但不是必须）实现默认方法；
- 如果实现接口的类实现了默认方法，那么使用这个类中的实现；
- 如果找不到其他实现，就使用默认实现。

`sort()` 方法是默认方法的一例，JDK 8 把它添加到 `java.util.List` 接口中，定义如下：

```java
// 句法<E>是Java编写泛型的方式,详情参见下一节
// 如果不熟悉泛型,暂且忽略这个句法
```

```
interface List<E> {
  // 省略了其他成员

  public default void sort(Comparator<? super E> c) {
    Collections.<E>sort(this, c);
  }
}
```

因此，从 Java 8 开始，实现 `List` 接口的对象都有一个名为 `sort()` 的实例方法，使用合适的 `Comparator` 排序列表。因为返回类型是 `void`，所以我们猜测这是就地排序，而事实确实如此。

4.1.6 标记接口

有时，定义全空的接口很有用。类实现这种接口时只需在 `implements` 子句中列出这个接口，而不用实现任何方法。此时，这个类的任何实例都是这个接口的有效实例。Java 代码可以使用 `instanceof` 运算符检查实例是否属于这个接口，因此这种技术是为对象提供额外信息的有力方式。

`java.io.Serializable` 接口就是一种标记接口。实现 `Serializable` 接口的类告诉 `ObjectOutputStream` 类，这个类的实例可以安全地序列化。`java.util.RandomAccess` 也是标记接口：`java.util.List` 接口实现了这个接口，表明这个接口能快速随机访问列表中的元素。例如，`ArrayList` 类实现了 `RandomAccess` 接口，而 `LinkedList` 类没实现。注重随机访问操作性能的算法可以使用下述方式测试 `RandomAccess`：

```
// 排序任意长度的列表元素之前,我们或许想确认列表是否支持快速随机访问
// 如果不支持,先创建一个支持随机访问的副本再排序,速度可能更快
// 注意,使用java.util.Collections.sort()时不必这么做
List l = ...; // 随意一个列表
if (l.size() > 2 && !(l instanceof RandomAccess)) l = new ArrayList(l);
sortListInPlace(l);
```

后面会看到，Java 的类型系统和类型的名称联系紧密，这种方式叫作*名义类型*（nominal typing）。标记接口是个很好的例子，因为它除了名称什么都没有。

4.2 Java泛型

Java 平台的一大优势是它提供的标准库。标准库提供了大量有用的功能，特别是实现了健壮的通用数据结构。这些实现使用起来相当简单，而且文档编写良好。这些是 Java 集合库，第 8 章会使用大量篇幅介绍。更完整的介绍参阅 Maurice Naftalin 和 Philip Wadler 合著的 *Java Generics and Collections*（http://shop.oreilly.com/product/9780596527754.do，O'Reilly 出版）。

虽然这些库一直很有用，但在早期版本中有相当大的不足——数据结构（经常叫作*容器*）

完全隐藏了存储其中的数据类型。

> 数据隐藏和封装是面向对象编程的重要原则，但在这种情况下，容器的不透明会为开发者带来很多问题。

本节先说明这个问题，然后介绍*泛型*是如何解决这个问题并让 Java 开发者的生活更轻松的。

4.2.1　介绍泛型

如果想构建一个由 `Shape` 实例组成的集合，可以把这个集合保存在一个 `List` 对象中，如下所示：

```
List shapes = new ArrayList(); // 创建一个List对象,保存形状

// 指定中心点,创建一些形状,保存在这个列表中
shapes.add(new CenteredCircle(1.0, 1.0, 1.0));
// 这是合法的Java代码,但不是好的设计方式
shapes.add(new CenteredSquare(2.5, 2, 3));

// List::get()返回Object对象,所以要想得到CenteredCircle对象,必须校正
CenteredCircle c = (CentredCircle)shapes.get(0);

// 下面这行代码会导致运行时失败
CenteredCircle c = (CentredCircle)shapes.get(1);
```

上述代码有个问题，为了取回有用的形状对象形式，必须校正，因为 `List` 不知道其中的对象是什么类型。不仅如此，其实可以把不同类型的对象放在同一个容器中，一切都能正常运行，但是如果做了不合法的校正，程序就会崩溃。

我们真正需要的是一种知道所含元素类型的 `List`。这样，如果把不合法的参数传给 `List` 的方法，`javac` 就能检测到，导致编译出错，而不用等到运行时才发现问题。

为了解决这个问题，Java 提供了一种句法，指明某种类型是一个容器，这个容器中保存着其他引用类型的实例。容器中保存的*负载类型*（payload type）在尖括号中指定：

```
// 创建一个由CenteredCircle对象组成的List
List<CenteredCircle> shapes = new ArrayList<CenteredCircle>();

// 指定中心点,创建一些形状,保存在这个列表中
shapes.add(new CenteredCircle(1.0, 1.0, 1.0));

// 下面这行代码会导致编译出错
shapes.add(new CenteredSquare(2.5, 2, 3));
```

```
// List<CenteredCircle>::get()返回一个CenteredCircle对象,无需校正
CenteredCircle c = shapes.get(0);
```

这种句法能让编译器捕获大量不安全的代码，根本不能靠近运行时。当然，这正是静态类型系统的关键所在——使用编译时信息协助排除大量运行时问题。

容器类型一般叫作*泛型*（generic type），使用下述方式声明：

```
interface Box<T> {
  void box(T t);
  T unbox();
}
```

上述代码表明，`Box` 接口是通用结构，可以保存任意类型的负载。这不是一个完整的接口，更像是一系列接口的通用描述，每个接口对应的类型都能用在 `T` 的位置上。

4.2.2 泛型和类型参数

我们已经知道如何使用泛型增强程序的安全性——使用编译时信息避免简单的类型错误。本节深入介绍泛型的特性。

`<T>` 句法有个专门的名称——*类型参数*（type parameter）。因此，泛型还有一个名称——*参数化类型*（parameterized type）。这表明，容器类型（例如 `List`）由其他类型（负载类型）参数化。把类型写为 `Map<String, Integer>` 时，我们就为类型参数指定了具体的值。

定义有参数的类型时，要使用一种不对类型参数做任何假设的方式指定具体的值。所以 `List` 类型使用通用的方式 `List<E>` 声明，而且自始至终都使用类型参数 `E` 作占位符，代表程序员使用 `List` 数据结构时负载的真实类型。

类型参数始终代表引用类型。类型参数的值不能使用基本类型。

类型参数可以在方法的签名和主体中使用，就像是真正的类型一样，例如：

```
interface List<E> extends Collection<E> {
  boolean add(E e);
  E get(int index);
  // 其他方法省略了
}
```

注意，类型参数 `E` 既可以作为返回类型的参数，也可以作为方法参数类型的参数。我们不假设负载类型有任何具体的特性，只对一致性做了基本假设，即存入的类型和后来取回的

类型一致。

4.2.3　菱形句法

创建泛型的实例时，赋值语句的右侧会重复类型参数的值。一般情况下，这个信息是不必要的，因为编译器能推导出类型参数的值。在 Java 的现代版本中，可以使用*菱形句法*省略重复的类型值。

下面通过一个示例说明如何使用菱形句法，这个例子改自之前的示例：

```
// 使用菱形句法创建一个由CenteredCircle对象组成的List
List<CenteredCircle> shapes = new ArrayList<>();
```

对这种冗长的赋值语句来说，这是个小改进，能少输入几个字符。本章末尾介绍 lambda 表达式时会再次讨论类型推导。

4.2.4　类型擦除

4.1.5 节说过，Java 平台十分看重向后兼容性。Java 5 添加的泛型又是一个会导致向后兼容性问题的新语言特性。

问题的关键是，如何让类型系统既能使用旧的非泛型集合类又能使用新的泛型集合类。设计者选择的解决方式是使用校正：

```
List someThings = getSomeThings();
// 这种校正不安全,但我们知道someThings的内容确实是字符串
List<String> myStrings = (List<String>)someThings;
```

上述代码表明，作为类型，List 和 List<String> 是兼容的，至少在某种程度上是兼容的。Java 通过*类型擦除*实现这种兼容性。这表明，泛型的类型参数只在编译时可见——javac 会去掉类型参数，而且在字节码中不体现出来。[1]

> 非泛型的 List 一般叫作*原始类型*（raw type）。就算现在有泛型了，Java 也完全能处理类型的原始形式。不过，这么做几乎就表明代码的质量不高。

类型擦除机制扩大了 javac 和 JVM 使用的类型系统之间的区别，4.6 节会详细说明。

类型擦除还能禁止使用某些其他定义方式，如果没有这个机制，代码看起来是合法的。在下述代码中，我们想使用两个稍微不同的数据结构计算订单数量：

注 1：会保留泛型的一些细微踪迹，在运行时通过反射能看到。

```
// 不会编译
interface OrderCounter {
  // 把名称映射到由订单号组成的列表上
  int totalOrders(Map<String, List<String>> orders);

  // 把名称映射到目前已下订单的总数上
  int totalOrders(Map<String, Integer> orders);
}
```

看起来这是完全合法的 Java 代码，但其实无法编译。问题是，这两个方法虽然看起来像是常规的重载，但擦除类型后，两个方法的签名都变成了：

```
int totalOrders(Map);
```

擦除类型后剩下的只有容器的原始类型，在这个例子中是 `Map`。运行时无法通过签名区分这两个方法，所以，Java 语言规范把这种句法列为不合法的句法。

4.2.5 通配符

参数化类型，例如 `ArrayList<T>`，不能实例化，即不能创建这种类型的实例。这是因为 `<T>` 是类型参数，只是真实类型的占位符。只有为类型参数提供具体的值之后（例如 `ArrayList<String>`），这个类型才算完整，才能创建这种类型的对象。

如果编译时不知道我们要使用什么类型，就会出现问题。幸好，Java 类型系统能调解这种问题。在 Java 中，有“未知类型”这个明确的概念，使用 `<?>` 表示。这是一种最简单的 Java *通配符类型*（wildcard type）。

涉及未知类型的表达式可以这么写：

```
ArrayList<?> mysteryList = unknownList();
Object o = mysteryList.get(0);
```

这是完全有效的 Java 代码——`ArrayList<?>` 和 `ArrayList<T>` 不一样，前者是变量可以使用的完整类型。我们对 `mysteryList` 的负载类型一无所知，但这对我们的代码来说不是问题。在用户的代码中使用未知类型时，有些限制。例如，下面的代码不会编译：

```
// 不会编译
mysteryList.add(new Object());
```

原因很简单，我们不知道 `mysteryList` 的负载类型。例如，如果 `mysteryList` 是 `ArrayList<String>` 类型的实例，那么就不能把 `Object` 对象存入其中。

始终可以存入容器的唯一一个值是 `null`，因为我们知道 `null` 可能是任何引用类型的值。但这没什么用，因此，Java 语言规范禁止实例化负载为未知类型的容器类型，例如：

```
// 不会编译
List<?> unknowns = new ArrayList<?>();
```

使用未知类型时有必要问这么一个问题："List<String> 是 List<Object> 的子类型吗？"即，能否编写如下的代码：

```
// 这么写合法吗？
List<Object> objects = new ArrayList<String>();
```

乍看起来，这么写完全可行，因为 String 是 Object 的子类，所以我们知道集合中的任何一个 String 类型元素都是有效的 Object 对象。不过，看看下述代码：

```
// 这么写合法吗？
List<Object> objects = new ArrayList<String>();

// 如果合法,那下面这行代码呢？
objects.add(new Object());
```

既然 objects 的类型声明为 List<Object>，那么就能把 Object 实例存入其中。然而，这个实例保存的是字符串，尝试存入的 Object 对象与其不兼容，因此这个操作在运行时会失败。

上述问题的答案是，虽然下述代码是合法的（因为 String 类继承 Object 类）：

```
Object o = new String("X");
```

但并不意味着泛型容器类型对应的语句也合法：

```
// 不会编译
List<Object> objects = new ArrayList<String>();
```

换种方式说，即 List<String> 不是 List<Object> 的子类型。如果想让容器的类型具有父子关系，需要使用未知类型：

```
// 完全合法
List<?> objects = new ArrayList<String>();
```

这表明，List<String> 是 List<?> 的子类型。不过，使用上述这种赋值语句时，会丢失一些类型信息。例如，get() 方法的返回类型现在实际上是 Object。还要注意，不管 T 的值是什么，List<?> 都不是 List<T> 的子类型。

未知类型有时会让开发者困惑，问些引人深思的问题，例如："为什么不使用 Object 代替未知类型？"不过，如前文所述，为了实现泛型之间的父子关系，必须有一种表示未知类型的方式。

1. 受限通配符

其实，Java 的通配符类型不止有未知类型一种，还有*受限通配符*（bounded wildcard）这个概念。受限通配符也叫*类型参数约束条件*，作用是限制类型参数的值能使用哪些类型。

受限通配符描述几乎不知道是什么类型的未知类型的层次结构，其实想表达的是这种意思：“我不知道到底是什么类型，但我知道这种类型实现了 `List` 接口。”在类型参数中，这句话表达的意思可以写成 `? extends List`。这为程序员提供了一线希望，至少知道可以使用的类型要满足什么条件，而不是对类型一无所知。

> 不管限定使用的类型是类还是接口，都要使用 `extends` 关键字。

这是*类型变体*（type variance）的一个示例。类型变体是容器类型之间的继承关系和负载类型的继承关系有所关联的理论基础。

- *类型协变*

  这表示容器类型之间和负载类型之间具有相同的关系。这种关系通过 `extends` 关键字表示。

- *类型逆变*

  这表示容器类型之间和负载类型之间具有相反的关系。这种关系通过 `super` 关键字表示。

容器类型作为类型的制造者或使用者时会体现这些原则。例如，如果 `Cat` 类扩展 `Pet` 类，那么 `List<Cat>` 是 `List<? extends Pet>` 的子类型。这里，`List` 是 `Cat` 对象的*制造者*，应该使用关键字 `extends`。

如果容器类型只是某种类型实例的*使用者*，就应该使用 `super` 关键字。

> Joshua Bloch 把这种用法总结成“Producer Extends, Consumer Super”原则（简称 PECS，“制造者使用 `extends`，使用者使用 `super`”）。

第 8 章会看到，Java 集合库大量使用了协变和逆变。大量使用这两种变体的目的是确保泛型“做正确的事”，以及表现出的行为不会让开发者诧异。

2. 数组协变

在早期的 Java 版本中，集合库还没有出现，容器类型的类型变体问题在 Java 的数组中也有体现。没有类型变体，即使 `sort()` 这样简单的方法也很难使用有效的方式编写：

```
Arrays.sort(Object[] a);
```

基于这个原因，Java 的数组可以协变——尽管这么做让静态类型系统暴露出了缺陷，但在 Java 平台的早期阶段仍是必要之恶：

```
// 这样写完全合法
String[] words = {"Hello World!"};
Object[] objects = words;

// 哦,天哪,运行时错误
objects[0] = new Integer(42);
```

最近对现代开源项目的研究表明，数组协变极少使用，几乎可以断定为编程语言的设计缺陷。[2] 因此，编写新代码时，应该避免使用数组协变。

3. 泛型方法

*泛型方法*是参数可以使用任何引用类型实例的方法。

例如，下述方法模拟 C 语言中 ,（逗号）运算符的功能。这个运算符一般用来合并有副作用的表达式。

```
// 注意,这个类不是泛型类
public class Utils
  public static <T> T comma(T a, T b) {
    return a;
  }
}
```

虽然这个方法的定义中使用了类型参数，但所在的类不需要定义为泛型类。使用这种句法是为了表明这个方法可以自由使用，而且返回类型和参数的类型一样。

4. 使用和设计泛型

使用 Java 的泛型时，有时要从两方面思考问题。

- *使用者*

  使用者要使用现有的泛型库，还要编写一些相对简单的泛型类。对使用者来说，要理解类型擦除的基本知识，因为如果不知道运行时对泛型的处理方式，会对几个 Java 句法感到困惑。

- *设计者*

  使用泛型开发新库时，设计者需要理解泛型的更多功能。规范中有一些难以理解的部分，例如要完全理解通配符和“capture-of”错误消息 [3] 等高级话题。

注 2：Raoul-Gabriel Urma and Janina Voigt,“Using the OpenJDK to Investigate Covariance in Java”, *Java Magazine* (May/June 2012):44–47.

注 3：指通配符类型导致的错误消息，例如 `set(int,capture of ?) in java.util.List<capture of ?> cannot be applied to (int,java.lang.Object)`。——译者注

泛型是 Java 语言规范中最难理解的部分之一，潜藏很多极端情况，并不需要每个开发者都完全理解，至少初次接触 Java 的类型系统时没必要。

4.2.6　编译时和运行时类型

假设有如下的代码片段：

```
List<String> l = new ArrayList<>();
System.out.println(l);
```

我们可以问这个问题：`l` 是什么类型？答案取决于在编译时（即 `javac` 看到的类型）还是运行时（JVM 看到的类型）问这个问题。

`javac` 把 `l` 看成 `List-of-String` 类型，而且会用这个类型信息仔细检查句法错误，例如不能使用 `add()` 方法添加不合法的类型。

而 JVM 把 `l` 看成 `ArrayList` 类型的对象，这一点可以从 `println()` 语句的输出中证实。因为要擦除类型，所以运行时 `l` 是原始类型。

因此，编译时和运行时的类型稍微有些不同。某种程度上，这个不同点是，运行时类型既比编译时类型精确，又没有编译时类型精确。

运行时类型没有编译时类型精确，因为没有负载类型的信息——这个信息被擦除了，得到的运行时类型只是原始类型。

编译时类型没有运行时类型精确，因为我们不知道 `l` 的具体类型到底是什么，只知道是一种和 `List` 兼容的类型。

4.3　枚举和注解

Java 有两种特殊形式的类和接口，在类型系统中扮演着特定的角色。这两种类型是*枚举类型*（enumerated type）和*注解类型*（annotation type），一般直接称为*枚举*和*注解*。

4.3.1　枚举

枚举是类的变种，功能有限，而且允许使用的值很少。

例如，假设我们想定义一个类型，表示三原色红绿蓝，而且希望这个类型只有这三个可以使用的值。我们可以使用 `enum` 关键字定义这个类型：

```
public enum PrimaryColor {
  // 实例列表末尾的分号是可选的
  RED, GREEN, BLUE
}
```

PrimaryColor 类型的实例可以按照静态字段的方式引用：PrimaryColor.RED、PrimaryColor.GREEN 和 PrimaryColor.BLUE。

> 在其他语言中，例如 C++，枚举一般使用整数常量实现，但 Java 采用的方式能提供更好的类型安全性和灵活性。例如，因为枚举是特殊的类，所以可以拥有成员，即字段和方法。如果字段或方法有主体，那么实例列表后面必须加上分号。

例如，假设我们要定义一个枚举，包含前几个正多边形（等边等角的形状），而且想为这些形状指定一些属性（在方法中指定）。我们可以使用接收一个参数的枚举实现这个需求，如下所示：

```
public enum RegularPolygon {
  // 有参数的枚举必须使用分号
  TRIANGLE(3), SQUARE(4), PENTAGON(5), HEXAGON(6);

  private Shape shape;

  public Shape getShape() {
    return shape;
  }

  private RegularPolygon(int sides) {
    switch (sides) {
      case 3:
        // 假设这些形状的构造方法接收的参数是边长和角度
        shape = new Triangle(1,1,1,60,60,60);
        break;
      case 4:
        shape = new Rectangle(1,1);
        break;
      case 5:
        shape = new Pentagon(1,1,1,1,1,108,108,108,108,108);
        break;
      case 6:
        shape = new Hexagon(1,1,1,1,1,1,120,120,120,120,120,120);
        break;
    }
  }
}
```

参数（在这个例子中只有一个参数）传入构造方法，创建单个枚举实例。因为枚举实例由 Java 运行时创建，而且在外部不能实例化，所以把构造方法声明为私有方法。

枚举有些特殊的特性：

- 都（隐式）扩展 java.lang.Enum 类；
- 不能泛型化；

- 可以实现接口；
- 不能被扩展；
- 如果枚举中的所有值都有实现主体，那么只能定义为抽象方法；
- 只能有一个私有（或使用默认访问权限）的构造方法。

4.3.2 注解

注解是一种特殊的接口。如名称所示，其作用是注解 Java 程序的某个部分。

例如 `@Override` 注解。在前面的一些示例中你可能见到过这个注解，想知道它有什么作用。

简单来说，什么作用也没有。这个答案或许会让你感到诧异。

说得稍微详细（也轻率）一点儿，注解没有直接作用，`@Override` 只是为注解的方法提供额外的信息，注明这个方法覆盖了超类中的方法。

注解能为编译器和集成开发环境（Integrated Development Environment，IDE）提供有用的提示。如果开发者把方法的名称拼写错了，而这个方法本来是要覆盖超类的方法，那么，在这个名称拼错的方法上使用 `@Override` 注解，可以提醒编译器什么地方出错了。

注解不能改变程序的语义，只能提供可选的元信息。严格说来，这意味着注解不能影响程序的执行，只能为编译器和其他预执行阶段提供信息。

Java 平台在 `java.lang` 中定义了为数不多的基本注解。一开始只支持 `@Override`、`@Deprecated` 和 `@SuppressWarnings`，这三个注解的作用分别是：注明方法是覆盖的，注明方法废弃了，以及静默编译器生成的警告。

后来，Java 7 增加了 `@SafeVarargs`（为变长参数方法提供增强的警告静默功能），Java 8 增加了 `@FunctionalInterface`。`@FunctionalInterface` 表示接口可以用作 lambda 表达式的目标。这是个很有用的标记注解，但不是必须使用的，后文会介绍。

和普通的接口相比，注解有些特殊的特性：

- 都（隐式）扩展 `java.lang.annotation.Annotation` 接口；
- 不能泛型化；
- 不能扩展其他接口；
- 只能定义没有参数的方法；
- 不能定义会抛出异常的方法；
- 方法的返回类型有限制；
- 方法可以有一个默认返回值。

4.3.3 自定义注解

自定义在自己的代码中使用的注解类型没什么难度。开发者可以使用 @interface 关键字定义新的注解类型，与定义类和接口的方式差不多。

> 自定义注解的关键是使用“元注解”。元注解是特殊的注解，用来注解新（自定义）注解类型的定义。

元注解在 java.lang.annotation 包中定义。开发者使用元注解指定新的注解类型能在哪里使用，以及编译器和运行时如何处理注解。

创建新的注解类型时，必须使用两个基本的元注解——@Target 和 @Retention。这两个注解接受的值都在枚举中定义。

@Target 元注解指明自定义的新注解能在 Java 源码的什么地方使用。可用的值在枚举 ElementType 中定义，包括：TYPE、FIELD、METHOD、PARAMETER、CONSTRUCTOR、LOCAL_VARIABLE、ANNOTATION_TYPE、PACKAGE、TYPE_PARAMETER 和 TYPE_USE。

另一个元注解 @Retention 指明 javac 和 Java 运行时如何处理自定义的注解类型。可使用的值有三个，在枚举 RetentionPolicy 中定义。

- SOURCE

  使用这个保留原则的注解，编译时会被 javac 丢弃。

- CLASS

  表示注解会出现在类文件中，但运行时 JVM 无法访问。这个值很少使用，但有时会在 JVM 字节码的离线分析工具中见到。

- RUNTIME

  表示用户的代码在运行时（使用反射）能访问这个注解。

下面看个示例。这是个简单的注解，名为 @Nickname。开发者使用这个注解为方法指定一个昵称，运行时使用反射可以找到这个方法。

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Nickname {
    String[] value() default {};
}
```

定义注解要做的就这么多——先指明注解能出现在哪里，然后是保留原则，最后是注解的

名称。因为我们要给一个方法起昵称，所以还要在这个注解上定义一个方法。撇开这一点，自定义注解是个非常简单的任务。

除了两个基本的元注解之外，还有两个元注解：@Inherited 和 @Documented。实际使用中很少见到这两个注解，它们的详细说明参见 Java 平台的文档。

4.3.4 类型注解

Java 8 为枚举 ElementType 添加了两个新值：TYPE_PARAMETER 和 TYPE_USE。添加这两个值后，注解能在以前不能出现的地方使用了，例如使用类型的所有地方。现在，开发者可以编写如下的代码：

```
@NotNull String safeString = getMyString();
```

@NotNull 传达的额外类型信息可在特殊的类型检查程序中使用，用于检测问题（对这个例子来说，可能抛出 NullPointerException 异常），还能执行额外的静态分析。Java 8 基本版自带了一些插入式类型检查程序，还提供了一个框架，开发者和库的作者可以使用这个框架自己编写类型检查程序。

本节介绍了 Java 的枚举和注解类型。下面介绍 Java 类型系统的另一个重要组成部分——嵌套类型。

4.4 嵌套类型

目前，书中见到的类、接口和枚举类型都定义为*顶层类型*。也就是说，都是包的直接成员，独立于其他类型。不过，类型还可以嵌套在其他类型中定义。这种类型是*嵌套类型*（nested type），一般称为“内部类”，是 Java 语言的一个强大功能。

嵌套类型有两个独立的目的，但都和封装有关。

- 如果某个类型需要特别深入地访问另一个类型的内部实现，可以嵌套定义这个类型。作为成员类型的嵌套类型，其访问方式与访问成员变量和方法的方式一样，而且能打破封装的规则。
- 某个类型可能只在特定的情况下需要使用，而且只在非常小的代码区域使用。这个类型应该密封在一个小范围内，因为它其实是实现细节的一部分，应该封装在一个系统的其他部分无法接触到的地方。

嵌套类型也可以理解为通过某种方式和其他类型绑定在一起的类型，不作为完全独立的实体真实存在。类型能通过四种不同的方式嵌套在其他类型中。

- *静态成员类型*

  静态成员类型是定义为其他类型静态成员的类型。嵌套的接口、枚举和注解始终都是静态成员类型（就算不使用 `static` 关键字也是）。

- *非静态成员类*

  “非静态成员类型”就是没使用 `static` 声明的成员类型。只有类才能作为非静态成员类型。

- *局部类*

  局部类是在 Java 代码块中定义的类，只在这个块中可见。接口、枚举和注解不能定义为局部类型。

- *匿名类*

  匿名类是一种局部类，但对 Java 语言来说没有有意义的名称。接口、枚举和注解不能定义为匿名类型。

“嵌套类型”这个术语虽然正确且准确，但开发者并没有普遍使用，大多数 Java 程序员使用的是一个意义模糊的术语——“内部类”。根据语境的不同，这个术语可以指代非静态成员类、局部类或匿名类，但不能指代静态成员类型，因此使用“内部类”这个术语时无法区分指代的是哪种嵌套类型。

虽然表示各种嵌套类型的术语并不总是那么明确，但幸运的是，从语境中一般都能确定应该使用哪种句法。

下面详细介绍这四种嵌套类型。每种类型都用单独的一节介绍其特性，使用时的限制，以及专用的 Java 句法。介绍完这四种嵌套类型之后，还有一节说明嵌套类型的运作方式。

4.4.1　静态成员类型

*静态成员类型*和普通的顶层类型很像，但为了方便，把它嵌套在另一个类型的命名空间中。静态成员类型有如下的基本特性。

- 静态成员类型类似于类的其他静态成员：静态字段和静态方法；
- 静态成员类型和所在类的任何实例都不关联（即没有 `this` 对象）；
- 静态成员类型只能访问所在类的静态成员；
- 静态成员类型能访问所在类型中的所有静态成员（包括其他静态成员类型）；
- 不管使不使用 `static` 关键字，嵌套的接口、枚举和注解都隐式声明为静态类型；
- 接口或注解中的嵌套类型也都隐式声明为静态类型；
- 静态成员类型可以在顶层类型中定义，也可以嵌入任何深度的其他静态成员类型中；
- 静态成员类型不能在其他嵌套类型中定义。

下面通过一个简单的例子介绍静态成员类型的句法。示例 4-1 定义了一个辅助接口，是所在类的静态成员。这个示例还展示了如何在定义这个接口的类内部以及外部的类中使用这个接口。注意，在外部类中要使用这个接口在层次结构中的名称。

示例 4-1：定义和使用一个静态成员接口

```
// 用链表实现堆栈的类
public class LinkedStack {

    // 这个静态成员接口定义如何链接对象
    // static关键字是可选的,因为所有嵌套接口都是静态类型
    static interface Linkable {
        public Linkable getNext();
        public void setNext(Linkable node);
    }

    // 链表的头节点是一个Linkable对象
    Linkable head;

    // 方法主体省略了
    public void push(Linkable node) { ... }

    public Object pop() { ... }
}

// 这个类实现前面定义的静态成员接口
class LinkableInteger implements LinkedStack.Linkable {
    // 这里是节点的数据和构造方法
    int i;
    public LinkableInteger(int i) { this.i = i; }

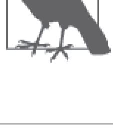
    // 这些是实现这个接口所需的数据和方法
    LinkedStack.Linkable next;

    public LinkedStack.Linkable getNext() { return next; }

    public void setNext(LinkedStack.Linkable node) { next = node; }
}
```

静态成员类型的特性

静态成员类型能访问所在类型中的所有静态成员，包括私有成员。反过来也成立：所在类型的方法能访问静态成员类型中的所有成员，包括私有成员。静态成员类型甚至能访问任何其他静态成员类型中的所有成员，包括这些类型的私有成员。静态成员类型使用其他静态成员时，无需使用所在类型的名称限定成员的名称。



静态成员类型不能和任何一个外层类同名。而且，静态成员类型只能在顶层类型和其他静态成员类型中定义，也就是说，静态成员类型不能在任何成员类、局部类或匿名类中定义。

顶层类型可以声明为 public 或对包私有（即声明时没使用 public 关键字）。但是把顶层类型声明为 private 或 protected 都没什么意义——protected 和对包私有其实一样，而任何其他类型都不能访问声明为 private 的顶层类。

然而，静态成员类型是一种成员，因此所在类型中的成员能使用的访问控制修饰符，静态成员类型都能使用。这些修饰符对静态成员类型来说，作用与用在类型的其他成员上一样。前面说过，接口（和注解）的所有成员都隐式声明为 public，所以嵌套在接口或注解类型中的静态成员类型不能声明为 protected 或 private。

例如，在示例 4-1 中，Linkable 接口声明为 public，因此任何想存储 LinkedStack 对象的类都可以实现这个接口。

在所在类外部，静态成员类型的名称由外层类型的名称和内层类型的名称组成（例如，LinkedStack.Link able）。

大多数情况下，这种句法有助于提醒内层类和所在的类型有内在联系。不过，Java 语言允许使用 import 指令直接或间接导入静态成员类型：

```
import pkg.LinkedStack.Linkable;  // 导入指定的嵌套类型
// 导入LinkedStack中的所有嵌套类型
import pkg.LinkedStack.*;
```

导入后，引用嵌套类型时就不用包含外层类型的名称了（例如，可以直接使用 Linkable）。

> 还可以使用 import static 指令导入静态成员类型。import 和 import static 的详细说明参阅 2.10 节。

但是，导入嵌套类型模糊了这个类型和外层类型之间的关系，而这种关系往往很重要，因此很少这么做。

4.4.2　非静态成员类

非静态成员类声明为外层类或枚举类型的成员，而且不使用 static 关键字：

- 如果把静态成员类型比作类字段或类方法，那么非静态成员类可以比作实例字段或实例方法；
- 只有类才能作为非静态成员类型；
- 一个非静态成员类的实例始终关联一个外层类型的实例；
- 非静态成员类的代码能访问外层类型的所有字段和方法（静态和非静态的都能访问）；
- 为了让非静态成员类访问外层实例，Java 提供了几个专用的句法。

示例 4-2 展示了如何定义和使用成员类。这个示例以前面定义的 LinkedStack 类为基础，增加了 iterator() 方法。这个方法返回一个实现 java.util.Iterator 接口的实例，枚举栈中的元素。实现这个接口的类定义为一个成员类。

示例 4-2：通过成员类实现的迭代器

```
import java.util.Iterator;

public class LinkedStack {

    // 静态成员接口
    public interface Linkable {
        public Linkable getNext();
        public void setNext(Linkable node);
    }

    // 链表的头节点
    private Linkable head;

    // 方法主体省略了
    public void push(Linkable node) { ... }
    public Linkable pop() { ... }

    // 这个方法返回一个Iterator对象,供LinkedStack类使用
    public Iterator<Linkable> iterator() { return new LinkedIterator(); }

    // 实现Iterator接口
    // 定义为一个非静态成员类
    protected class LinkedIterator implements Iterator<Linkable> {
        Linkable current;

        // 构造方法用到了外层类的一个私有字段
        public LinkedIterator() { current = head; }

        // 下面三个方法由Iterator接口定义
        public boolean hasNext() { return current != null; }

        public Linkable next() {
            if (current == null)
              throw new java.util.NoSuchElementException();
            Linkable value = current;
            current = current.getNext();
            return value;
        }

        public void remove() { throw new UnsupportedOperationException(); }
    }
}
```

注意，LinkedIterator 类嵌套在 LinkedStack 类中。因为 LinkedIterator 是辅助类，只在 LinkedStack 类中使用，所以在离外层类很近的地方定义，能清晰地表达设计意图——介绍嵌套类型时说过这一点。

1. 成员类的特性

与实例字段和实例方法一样，非静态成员类的每个实例都和外层类的一个实例关联。也就是说，成员类的代码能访问外层类实例的所有实例字段和实例方法（以及静态成员），包括声明为 `private` 的实例成员。

这个重要的特性在示例 4-2 中已经体现出来了。下面再次列出构造方法 `LinkedStack.LinkedIterator()`：

```
public LinkedIterator() { current = head; }
```

这一行代码把内层类的 `current` 字段设为外层类中 `head` 字段的值。即便 `head` 是外层类的私有字段，也不影响这行代码的正常运行。

非静态成员类和类的任何成员一样，可以使用一个标准的访问控制修饰符。在示例 4-2 中，`LinkedIterator` 类声明为 `protected`，所以使用 `LinkedStack` 类的代码（不同包）不能访问 `LinkedIterator` 类，但是 `LinkedStack` 的子类可以访问。

2. 成员类的限制

成员类有两个重要的限制。

- 非静态成员类不能和任何外层类或包同名。这是一个重要的规则，但不适用于字段和方法。
- 非静态成员类不能包含任何静态字段、方法或类型，不过可以包含同时使用 `static` 和 `final` 声明的常量字段。

> 静态成员是顶层结构，不和任何特定的对象关联，而非静态成员类和外层类的实例关联。在成员类中定义顶层静态成员会让人困惑，因此禁止这么做。

3. 成员类的句法

成员类最重要的特性是可以访问外层对象的实例字段和方法。从示例 4-2 中的构造方法 `LinkedStack.LinkedIterator()` 可以看出这一点：

```
public LinkedIterator() { current = head; }
```

在这个示例中，`head` 字段是外层 `LinkedStack` 类的字段，我们把这个字段的值赋值给 `LinkedIterator` 类的 `current` 字段（`current` 是非静态成员类的一个成员）。

如果想使用 `this` 显式引用，就要使用一种特殊的句法，显式引用 `this` 对象表示的外层实例。例如，如果想在这个构造方法中显式引用，可以使用下述句法：

```
public LinkedIterator() { this.current = LinkedStack.this.head; }
```

这种句法的一般形式是 classname.this，其中 classname 是外层类的名称。注意，成员类中可以包含成员类，嵌套的层级不限。然而，因为成员类不能和任何外层类同名，所以，在 this 前面使用外层类的名称是引用任何外层实例最好的通用方式。

> 仅当引用的外层类成员被成员类的同名成员遮盖时才必须使用这种特殊的句法。

4. 作用域和继承关系比较

我们注意到，顶层类可以扩展成员类。介绍非静态成员类之后，对任何类来说，都必须考虑两种独立的层次结构。第一种是从超类到子类的*继承层次结构*，定义成员类继承的字段和方法。第二种是从外层类到内层类的*包含层次结构*，定义在成员类作用域中的字段和方法（因此成员类可以访问这些字段和方法）。

你要熟悉这两种层次结构的特性和经验法则：

- 这两种层次结构完全相互独立，一定不能混淆；
- 避免命名冲突，即超类的字段或方法不能和外层类的字段或方法同名；
- 如果出现了命名冲突，那么继承的字段或方法取代外层类的同名字段或方法；
- 继承的字段和方法作用域在继承它们的类中，会取代外层作用域中的同名字段和方法；
- 为了避免混淆继承层次结构和包含层次结构，包含层次结构不要太深；
- 如果类嵌套超过两层，可能导致更大的混乱；
- 如果类的继承层次结构很深（即有很多祖先），可以考虑不把它定义为非静态成员类，而是定义为顶层类。

4.4.3　局部类

*局部类*在一个 Java 代码块中声明，不是类的成员。只有类才能局部定义，接口、枚举类型和注解类型都必须是顶层类型或静态成员类型。局部类往往在方法中定义，但也可以在类的静态初始化程序或实例初始化程序中定义。

因为所有 Java 代码块都在类中，所以局部类都嵌套在外层类中。因此，局部类和成员类有很多共同的特性。局部类往往更适合看成完全不同的嵌套类型。

第 5 章会详细说明什么时候适合使用局部类，什么时候适合使用 lambda 表达式。

局部类的典型特征是局部存在于一个代码块中。和局部变量一样，局部类只在定义它的块中有效。示例 4-3 修改 `LinkedStack` 类的 `iterator()` 方法，把 `LinkedIterator` 类从成员类改成局部类。

这样修改之后，我们把 `LinkedIterator` 类的定义移到离使用它更近的位置，希望更进一步提升代码的清晰度。简单起见，示例 4-3 只列出了 `iterator()` 方法，没有写出包含它的整个 `LinkedStack` 类。

示例 4-3：定义和使用一个局部类

```
// 这个方法返回一个Iterator对象,供LinkedStack类使用
public Iterator<Linkable> Iterator() {
    // 把LinkedIterator定义为局部类
    class LinkedIterator implements Iterator<Linkable> {
        Linkable current;

        // 构造方法用到了外层类的一个私有字段
        public LinkedIterator() { current = head; }

        // 下面三个方法由Iterator接口定义
        public boolean hasNext() { return current != null; }

        public Linkable next() {
            if (current == null)
              throw new java.util.NoSuchElementException();
            Linkable value = current;
            current = current.getNext();
            return value;
        }

        public void remove() { throw new UnsupportedOperationException(); }
    }

    // 创建并返回前面定义的类的一个实例
    return new LinkedIterator();
}
```

1. 局部类的特性

局部类有如下两个有趣的特性：

- 和成员类一样，局部类和外层实例关联，而且能访问外层类的任何成员，包括私有成员；
- 除了能访问外层类定义的字段之外，局部类还能访问局部方法的作用域中声明为 `final` 的任何局部变量、方法参数和异常参数。

2. 局部类的限制

局部类有如下限制。

- 局部类的名称只存在于定义它的块中，在块的外部不能使用。(但是要注意，在类的作用域中创建的局部类实例,在这个作用域之外仍能使用。稍后本节会详细说明这种情况。)
- 局部类不能声明为 `public`、`protected`、`private` 或 `static`。
- 与成员类的原因一样，局部类不能包含静态字段、方法或类。唯一的例外是同时使用 `static` 和 `final` 声明的常量。
- 接口、枚举类型和注解类型不能局部定义。
- 局部类和成员类一样，不能与任何外层类同名。
- 前面说过，局部类能使用同一个作用域中的局部变量、方法参数和异常参数，但这些变量或参数必须声明为 `final`。这是因为，局部类实例的生命周期可能比定义它的方法的执行时间长很多。

> 局部类用到的每个局部变量都有一个私有内部副本（这些副本由 `javac` 自动生成）。只有把局部变量声明为 `final` 才能保证局部变量和私有副本始终保持一致。

3. 局部类的作用域

介绍非静态成员类时，我们知道，成员类能访问继承自超类的任何成员以及外层类定义的任何成员。这对局部类来说也成立，但局部类还能访问声明为 `final` 的局部变量和参数。示例 4-4 展示了局部变量能访问的不同字段和变量种类：

示例 4-4：局部类能访问的字段和变量

```
class A { protected char a = 'a'; }
class B { protected char b = 'b'; }

public class C extends A {
  private char c = 'c';              // 私有字段,对局部类可见
  public static char d = 'd';
  public void createLocalObject(final char e)
  {
    final char f = 'f';
    int i = 0;                  // i没声明为final,局部类不能使用
    class Local extends B
    {
      char g = 'g';
      public void printVars()
      {
        // 这个类能访问所有下述字段和变量
        System.out.println(g);  // (this.g) g是这个类的字段
        System.out.println(f);  // f是声明为final的局部变量
        System.out.println(e);  // e是声明为final的局部参数
        System.out.println(d);  // (C.this.d) d是外层类的字段
```

```
      System.out.println(c);  // (C.this.c) c是外层类的字段
      System.out.println(b);  // b是这个类继承的字段
      System.out.println(a);  // a是外层类继承的字段
    }
  }
  Local l = new Local();          // 创建局部类的实例
  l.printVars();                  // 然后在这个实例上调用printVars()方法
 }
}
```

4.4.4 词法作用域和局部变量

局部变量在一个代码块中定义，这个代码块是这个变量的*作用域*，在这个作用域之外无法访问这个局部变量，局部变量也不复存在。花括号划定块的边界，花括号中的任何代码都能使用这个块中定义的局部变量。

这种作用域是*词法作用域*，定义变量能在哪一块源码种使用。程序员一般可以把这种作用域理解为*暂时*存在的事物，而不能认为局部变量的存在时间是从 JVM 开始执行代码块开始，到退出代码块为止。像这样理解局部变量和它的作用域一般是合理的。

但是，局部类的出现把这个局面搅乱了。注意，局部类的实例可能在 JVM 退出定义这个局部类的代码块后依然存在，这就是原因。

> 也就是说，如果创建了局部类的一个实例，那么，JVM 执行完定义这个类的代码块后，实例不会自动消失。因此，即便这个类在局部定义，但这个类的实例能跳出定义它的地方。

这可能会导致一些效果，让某些初次接触的开发者惊讶。这是因为，局部类能使用局部变量，而且会从不复存在的词法作用域中创建变量值的副本。这一点从下述代码种可以看出：

```
public class Weird {
  // 静态成员接口,下面会用到
  public static interface IntHolder { public int getValue(); }

  public static void main(String[] args) {
    IntHolder[] holders = new IntHolder[10];
    for(int i = 0; i < 10; i++) {
      final int fi = i;

      // 局部类
      class MyIntHolder implements IntHolder {
        // 使用前面定义的final变量
        public int getValue() { return fi; }
      }
      holders[i] = new MyIntHolder();
```

```
    }

    // 局部类不在作用域中了,因此不能使用
    // 但是在数组中保存有这个类的10个有效实例
    // 局部变量fi现在已经不在作用域中了
    // 但仍然在那10个对象getValue()方法的作用域中
    // 因此,可以在每个对象上调用getValue()方法,打印fi的值
    // 下述代码打印数字0到9
    for(int i = 0; i < 10; i++) {
      System.out.println(holders[i].getValue());
    }
  }
}
```

为了理解这段代码，要记住一点，局部类中方法的词法作用域与解释器进出定义局部类的代码块没有任何联系。

局部类的各个实例用到的每个 `final` 局部变量，都会自动创建一个私有副本，因此，得到的效果是，创建实例时，这个实例拥有一个所在作用域的私有副本。

> 局部类 `MyIntHolder` 有时也叫闭包（closure）。用更一般的 Java 术语来说，闭包是一个对象，它保存作用域的状态，并让这个作用域在后面可以继续使用。

在某些编程风格中闭包是有用的。不同的编程语言使用不同的方式定义和实现闭包，Java 通过局部类、匿名类和 lambda 表达式实现闭包。

4.4.5 匿名类

*匿名类*是没有名称的局部类，使用 `new` 运算符在一个简洁的表达式中定义和实例化。局部类是 Java 代码块中的一个语句，而匿名类是一个表达式，因此可以包含在大型表达式中，例如方法调用表达式。

> 为了完整介绍嵌套类，这里涵盖了匿名类，但在 Java 8 之后，大多数情况下都把匿名类换成了 lambda 表达式（参见本章的小结）。

示例 4-5 在 `LinkedStack` 类的 `iterator()` 方法中使用匿名类实现 `LinkedIterator` 类。和示例 4-4 对比一下，示例 4-4 使用局部类实现了同一个类。

示例 4-5：使用匿名类实现的枚举功能

```
public Iterator<Linkable> iterator() {
```

```
        // 匿名类在return语句中定义
        return new Iterator<Linkable>() {
            Linkable current;
            // 把构造方法换成实例初始化程序
            { current = head; }

            // 下面三个方法由Iterator接口定义
            public boolean hasNext() { return current != null; }
            public Linkable next() {
                if (current == null)
                  throw new java.util.NoSuchElementException();
                Linkable value = current;
                current = current.getNext();
                return value;
            }
            public void remove() { throw new UnsupportedOperationException(); }
        };  // 注意,需要使用分号,结束return语句
    }
```

可以看出，定义匿名类和创建这个类的实例使用 new 关键字，后面跟着某个类的名称和放在花括号里的类主体。如果 new 关键字后面是一个类的名称，那么这个匿名类是指定类的子类。如果 new 关键字后面是一个接口的名称，如前面的示例所示，那么这个匿名类实现指定的接口，并且扩展 Object 类。

> 匿名类使用的句法无法指定 extends 子句和 implements 子句，也不能为这个类指定名称。

因为匿名类没有名称，所以不能在类主体中定义构造方法。这是匿名类的一个基本限制。定义匿名类时，在父类后面的括号中指定的参数，会隐式传给父类的构造方法。匿名类一般用于创建构造方法不接受任何参数的简单类的子类，所以，在定义匿名类的句法中，括号经常都是空的。前面示例中的匿名类实现一个接口并扩展 Object 类。因为构造方法 Object() 不接受参数，所以括号是空的。

匿名类的限制

匿名类就是一种局部类，所以二者的限制一样。除了使用 static final 声明的常量之外，匿名类不能定义任何静态字段、方法和类。接口、枚举类型和注解类型不能匿名定义。而且，和局部类一样，匿名类不能声明为 public、private、protected 或 static。

定义匿名类的句法既定义了这个类也实例化了这个类。如果每次执行外层块时创建的实例不止一个，那么就不能用匿名类代替局部类。

因为匿名类没有名称，所以无法为匿名类定义构造方法。如果类需要构造方法，必须使用局部类。不过，经常可以使用实例初始化程序代替构造方法。

虽然实例初始化程序（3.3.4 节介绍过）不仅限于在匿名类中使用，但就是为了这个目的才把这种功能引入 Java 语言的。匿名类不能定义构造方法，所以只有一个默认构造方法。使用实例初始化程序可以打破匿名类不能定义构造方法这个限制。

4.4.6 嵌套类型的运作方式

前面几节说明了这四种嵌套类型的特性和行为。对于嵌套类型，尤其只是为了使用，你要知道的就这么多。不过，理解嵌套类型的运作方式后能更好地理解嵌套类型。

> 引入嵌套类型后，Java 虚拟机和 Java 类文件的格式并没有变化。对 Java 解释器而言，并没有所谓的嵌套类型，所有类都是普通的顶层类。

为了让嵌套类型看起来是在另一个类中定义的，Java 编译器会在它生成的类中插入隐藏字段、方法和构造方法参数。这些隐藏字段和方法经常称为*合成物*（synthetic）。

你可以使用反汇编程序 `javap`（第 13 章会介绍）反汇编某些嵌套类型的类文件，了解为了支持嵌套类型，编译器用了什么技巧。

为了实现嵌套类型，`javac` 把每个嵌套类型编译为单独的类文件，得到的其实是顶层类。编译得到的类文件使用特殊的命名约定，这些名称一般在用户的代码中无法创建。

在第一个 `LinkedStack` 类的示例（示例 4-1）中，定义了一个名为 `Linkable` 的静态成员接口。编译这个 `LinkedStack` 类时，编译器会生成两个类文件，第一个是预期的 LinkedStack.class。

不过，第二个类文件名为 LinkedStack$Linkable.class，其中，`$` 由 `javac` 自动插入。这个类文件中包含的就是静态成员接口 `Linkable` 的实现。

因为嵌套类型编译成普通的顶层类，所以不能直接访问外层类型中有特定权限的成员。因此，如果静态成员类型使用了外层类型的私有成员（或具有其他权限的成员），编译器会生成合成的访问方法（具有默认的包访问权限），然后把访问私有成员的表达式转换成调用合成方法的表达式。

这四种嵌套类型的类文件使用如下命名约定。

- *（静态或非静态）成员类型*
  根据 `EnclosingType$Member.class` 格式命名成员类型的类文件。

- *匿名类*

  因为匿名类没有名称，所以类文件的名称由实现细节决定。Oracle/OpenJDK 中的 `javac` 使用数字表示匿名类的名称（例如 `EnclosingType$1.class`）。

- *局部类*

  局部类的类文件综合使用前两种方式命名（例如 `EnclosingType$1Member.class`）。

下面简单介绍一些实现细节，看一下 `javac` 如何为每种嵌套类型提供所需的合成访问能力。

1. 非静态成员类的实现

非静态成员类的每个实例都和一个外层类的实例关联。为了实现这种关联，编译器为每个成员类定义了一个名为 `this$0` 的合成字段。这个字段的作用是保存一个外层实例的引用。

编译器为每个非静态成员类的构造方法提供了一个额外的参数，用于初始化这个字段。每次调用成员类的构造方法时，编译器都会自动把这个额外参数的值设为外层类的引用。

2. 局部类和匿名类的实现

局部类之所以能访问外层类的字段和方法，原因和非静态成员类一模一样：编译器把一个外层类的隐藏引用传入局部类的构造方法，并且把这个引用存储在编译器合成的一个私有字段中。和非静态成员类一样，局部类也能使用外层类的私有字段和方法，因为编译器会插入任何所需的访问器方法。

局部类和成员类的不同之处在于，局部类能访问所在块中的局部变量。不过这种能力有个重要的限制，即局部类只能访问声明为 `final` 的局部变量和参数。这个限制的原因从实现中可以清楚地看出来。

局部类之所以能使用局部变量，是因为 `javac` 自动为局部类创建了私有实例字段，保存局部类用到的各个局部变量的副本。

编译器还在局部类的构造方法中添加了隐藏的参数，初始化这些自动创建的私有字段。其实，局部类没有访问局部变量，真正访问的是局部变量的私有副本。如果在局部类外部能修改局部变量，就会导致不一致性。[4]

4.5　lambda表达式

Java 8 引入的功能中，最让人期盼的是 lambda 表达式。lambda 表达式以字面量的形式把少量代码直接写在程序中，而且让 Java 编程更符合函数式风格。

其实，lambda 表达式的很多功能都能使用嵌套类型通过回调和处理程序等模式实现，但使

注 4：第 6 章介绍内存管理和可变的状态时还会讨论这个话题。

用的句法总是非常冗长，尤其是，就算只需要在回调中编写一行代码，也要完整定义一个新类型。

第 2 章见过，lambda 表达式的句法是一个参数列表和方法主体，如下所示：

```
(p, q) -> { /* 方法主体 */ }
```

这种句法能通过一种十分紧凑的方式表示简单的方法，而且能很大程度上避免使用匿名类。

> 组成方法的各个部分，lambda 表达式几乎都有，不过显然，lambda 表达式没有名称。其实，有些开发者喜欢把 lambda 表达式当成“匿名方法”。

例如，`java.io.File` 类的 `list()` 方法。这个方法列出一个目录中的文件，但在返回列表之前，要把每个文件的名称传给 `FilenameFilter` 对象，而这个对象必须由你提供。`FilenameFilter` 对象用于接受或拒绝各个文件。

使用匿名类可以按照如下的方式定义一个 `FilenameFilter` 类，只列出文件名以 .java 结尾的文件：

```
File dir = new File("/src");      // 列出这个目录中的文件

// 现在调用list()方法,参数的值是一个使用匿名类实现的FilenameFilter
String[] filelist = dir.list(new FilenameFilter() {
  public boolean accept(File f, String s) {
     return s.endsWith(".java");
  }
});
```

使用 lambda 表达式，上述代码可以简化成：

```
File dir = new File("/src");     // 列出这个目录中的文件

String[] filelist = dir.list((f,s) -> { return s.endsWith(".java"); });
```

对目录中的每个文件来说，都会执行 lambda 表达式中的代码。如果这个方法的返回值是 `true`，对应的文件就会出现在输出中，即存入数组 `filelist` 中。

这种模式叫过滤器，即使用一个代码块测试容器中的元素是否匹配某个条件，并且只返回能通过条件的元素。过滤器是函数式编程的标准技术之一，稍后会详细说明。

4.5.1 转换lambda表达式

javac 遇到 lambda 表达式时会把它解释为一个方法的主体，这个方法具有特定的签名。不过，是哪个方法呢？

为了解决这个问题，javac 会查看周围的代码。lambda 表达式必须满足以下条件才算是合法的 Java 代码：

- lambda 表达式必须出现在期望使用接口类型实例的地方；
- 期望使用的接口类型必须只有一个强制方法；
- 这个强制方法的签名要完全匹配 lambda 表达式。

如果满足上述条件，编译器会创建一个类型，实现期望使用的接口，然后把 lambda 表达式的主体当作强制方法的实现。

说得稍微复杂一点儿，这么做是为了保持 Java 类型系统的名义（基于名称）纯粹性。也就是说，lambda 表达式会被转换成正确接口类型的实例。

有些开发者还喜欢使用“单一抽象方法”（Single Abstract Method，SAM）类型这个术语表示 lambda 表达式转换得到的接口类型。这表明，若想在 lambda 表达式机制中使用某个接口，这个接口必须只有一个非默认方法。

> 虽然 lambda 表达式和匿名类有很多相似之处，但 lambda 表达式并不只是匿名类的语法糖。其实，lambda 表达式使用方法句柄（第 11 章介绍）和一个特殊的新 JVM 字节码 invokedynamic 实现。

从上述讨论可以看出，Java 8 添加的 lambda 表达式经过精心设计，以适应 Java 现有的类型系统——这个系统十分注重名义类型。

4.5.2 方法引用

前面说过，可以把 lambda 表达式看成没有名称的方法。对下面的 lambda 表达式来说：

```
// 实际上这行代码可以写得更简短,因为有类型推导
(MyObject myObj) -> myObj.toString()
```

会自动转换成对 @FunctionalInterface 接口的实现，这个接口只有一个非默认方法，这个方法接受一个 MyObject 类型的参数，返回值类型为 String。不过，这里的样板代码太多，所以 Java 8 提供了一种句法，可以让这种 lambda 表达式更易于阅读和编写：

```
MyObject::toString
```

这种简写形式叫*方法引用*（method reference），使用现有的方法作为 lambda 表达式。方法引用就像是使用现有的方法，但会忽略方法的名称，所以能作为 lambda 表达式使用，而且能使用往常的方式自动转换。

4.5.3　函数式编程

Java 实质上是面向对象语言。不过，引入 lambda 表达式后，可以更轻易地编写符合函数式风格的代码。

> 关于*函数式语言*由什么组成，没有明确的定义，但至少有一个共识：函数式语言最起码要能把函数当成值，存入变量。

Java（从 1.1 版起）一直都能通过内部类表示函数，但句法很复杂，代码结构不清晰。lambda 表达式大大简化了这种句法，因此，越来越多的开发者会在 Java 代码中寻求使用函数式编程风格，这是很自然的，而且现在做起来也更容易。

Java 开发者初尝函数式编程时有可能会使用如下三个非常有用的基本习语。

- `map()`

  `map()` 用于列表和类似列表的容器。运作原理是，传入一个函数，应用于集合中的各个元素，得到一个新集合。新集合中保存的是在各个元素上执行函数后得到的结果。这意味着，`map()` 可能会把一种类型的集合转换成另一种类型的集合。

- `filter()`

  说明如何把匿名类实现的 `FilenameFilter` 换成 lambda 表达式实现时，见过使用 `filter()` 的示例。`filter()` 基于某种条件生成一个集合的子集。注意，在函数式编程中，一般会生成新集合，而不直接修改现有的集合。

- `reduce()`

  `reduce()` 有几种不同的形式，执行的是聚合运算，除了叫*化简*之外，还可以叫*合拢*、*累计*或*聚合*。基本原理是，提供一个初始值和聚合函数（或化简函数），然后在各个元素上执行这个化简函数，在化简函数遍历整个集合的过程中会得到一系列中间值（类似于“累积计数”），最后得到一个最终结果。

Java 完全支持这些重要的函数式习语（除此之外还有几个）。第 8 章会稍微深入地说明这些习语的实现方式，届时会介绍 Java 的数据结构和集合，以及抽象*流*。抽象流是实现这些习语的基础。

对 lambda 表达式的介绍到此结束，下面是一些注意事项。值得注意的是，最好把 Java 看成轻度支持函数式编程的语言。Java 不是专门的函数式语言，也不想变成函数式语言。Java 的某些特性决定了它不可能是函数式语言，具体而言有以下几点。

- Java 没有结构类型，因此没有“真正的”函数类型。每个 lambda 表达式都会自动转换成适当的名义类型。
- 类型擦除在函数式编程中会导致问题——高阶函数的类型安全性会丢失。
- Java 天生可改变（第 6 章会介绍）——一般认为，可变性是函数式语言极不需要的特性。

抛开这些，能轻易使用基本的函数式编程风格，尤其是 `map()`、`filter()` 和 `reduce()` 等习语，是 Java 社区向前迈出的一大步。这些习语非常有用，因此绝大多数 Java 开发者都不需要也不会错过纯正函数式语言提供的高级功能。

4.6　小结

了解 Java 的类型系统之后，我们对 Java 平台的数据类型有了清晰的全局性认识。Java 的类型系统具有如下特性。

- *名义*

  Java 类型的名称至关重要。Java 不允许使用其他语言支持的结构类型。

- *静态*

  所有 Java 变量在编译时都知道类型。

- *面向对象/命令式*

  Java 代码是面向对象的，所有代码都放在方法中，而方法放在类中。但是，Java 有基本类型，因此并非“一切皆对象”。

- *轻度函数式*

  Java 支持一些常用的函数式习语，但这只是为了给程序员提供方便，别无其他目的。

- *适度的类型推导*

  Java 为代码（即便是程序员新手写的代码）易读性做了优化，就算信息有重复，也倾向于明确表达意图。

- *极力向后兼容*

  Java 是一门主要针对商业应用的语言，所以向后兼容性和保护现有代码是关注的重点。

- *类型擦除*

  Java 允许使用参数化类型，但这些信息在运行时不可用。

这些年 Java 的类型系统一直在进化（尽管缓慢且谨慎），引入 lambda 表达式之后，变得和其他主流编程语言一样了。lambda 表达式和默认方法的引入是 Java 5 发布以来这个平台最大的变化，除此之外，还引入了泛型和注解等相关的革新。

默认方法是 Java 实现面向对象编程方式的重大转变，这或许是 Java 语言面世以来最大的一次转变。从 Java 8 开始，接口可以包含实现代码。这从根本上改变了 Java 的本质——以前 Java 只支持单一继承，现在则可以多重继承（只是表面上如此，其实并没有状态的多重继承）。

尽管做了这些革新，Java 的类型系统还是没有（也不打算比）Scala 和 Haskell 等语言的类型系统强大。Java 的类型系统偏向于简单、易读，为新人提供一个平缓的学习曲线。

Java 还从过去十年里出现的其他语言中获益良多。Scala 是一种静态类型语言，但通过使用类型推导，看起来很像是动态类型语言。Java 从中受到了启发，不过和其他语言的设计哲学不一样。

虽然等了好久 Java 才支持 lambda 表达式，但争论最终停歇了，Java 仍然是人们更好的选择。广大的普通 Java 程序员是否需要这些增加的功能（源于 Scala 等语言的高级且缺少名义的类型系统），以及 Java 8 的轻度函数式编程（例如，`map()`、`filter()` 和 `reduce()` 等）是否能满足大多数开发者的需求，还有待日后观察。这注定是一段有趣的旅程。

第 5 章

Java的面向对象设计

本章介绍如何使用 Java 的对象，涵盖 `Object` 类的重要方法、面向对象设计要略，以及异常处理机制的实现方式。本章会介绍一些设计模式，主要是解决软件设计中一些常见问题的最佳实践。本章末尾会介绍如何设计安全的程序，避免程序随时间的推移变得不一致。首先，我们要介绍 Java 调用和传值的约定，以及 Java 值的本性。

5.1 Java的值

Java 的值以及它们与类型系统的关系非常简单。Java 的值有两种类型——基本值和对象引用。

> 有些书把基本值称为“值类型”——把 Java 的对象引用当成值时，这个称呼会产生歧义。因此，只要涉及 Java 的八种非引用类型，我们都使用“基本值”这个术语。

只有这两种值才能赋值给变量。其实，值的一种定义方式是“可以赋值给变量或传入方法的东西”。C++ 和 C 语言程序员要注意，对象的内容不能赋值给变量，所以 Java 没有解除引用运算符或结构体。

基本值和对象引用的主要区别是，基本值不能修改——2 永远都是 2，而对象引用的内容一般都能修改——一般称这种修改为对象内容的变化（mutation）。

Java 试图简化一个经常会让 C++ 程序员困惑的概念，即“对象的内容”和“对象的引用”

之间的区别。但也不能完全忽视这个区别，因此，程序员要理解 Java 平台中引用值的运作方式。

> **Java 是“引用传递”语言吗？**
>
> Java“通过引用”处理对象，但不能把这种处理方式和“引用传递”（pass by reference）搞混了。“引用传递”是一个术语，用于描述某些编程语言中方法的调用方式。在引用传递语言中，值，甚至是基本值，不直接传入方法，而是把值的引用传入方法。因此，如果方法修改了参数，方法返回后这一变化仍然存在。就算是基本类型，也使用这种处理方式。
>
> Java 不会这么做，Java 是“值传递”语言。不过，如果传入的值是引用类型，那么实际传入的是引用副本。但是这和引用传递并不是一回事。如果 Java 是引用传递语言，把引用类型的值传入方法时，传入的应该是引用的引用。

事实上，Java 使用值传递，这一点很容易证明。如下述代码所示，就算调用了 `manipulate()` 方法，变量 `c` 保存的值也没有变化，还是引用一个半径为 2 的 `Circle` 对象。如果 Java 是引用传递语言，那么 `c` 保存的值应该是一个半径为 3 的 `Circle` 对象。

```
public void manipulate(Circle circle) {
    circle = new Circle(3);
}

Circle c = new Circle(2);
manipulate(c);
System.out.println("Radius: "+ c.getRadius());
```

如果我们谨慎对待这个区别，而且把对象引用当成 Java 的一种值，那么 Java 某些令人惊奇的其他功能就会显现出来。注意，有些旧资料对这一点的表述并不清晰。第 6 章介绍内存管理和垃圾回收机制时还会遇到 Java 值的这种特性。

5.2 java.lang.Object类的重要方法

前面说过，所有类都直接或间接扩展 `java.lang.Object` 类。这个类定义了很多有用的方法，而且你编写的类可以覆盖这些方法。示例 5-1 中的类覆盖了这些方法。这个示例之后的几节，说明各个方法的默认实现，以及为什么要覆盖。

这个示例使用了前一章介绍的多个类型系统的扩展功能。首先，这个示例使用参数化类型（或叫泛型）实现 `Comparable` 接口。其次，这个示例使用 `@Override` 注解，强调（并让编译器确认）某些方法覆盖了 `Object` 类中对应的方法。

示例 5-1：一个类，覆盖 `Object` 类的重要方法

```
// 这个类表示圆形,位置和半径都不能改变
```

```java
public class Circle implements Comparable<Circle> {
    // 这些字段存储圆心的坐标和圆的半径
    // 使用private是为了封装数据,使用final是为了禁止修改
    private final int x, y, r;

    // 基本构造方法:使用指定的值初始化字段
    public Circle(int x, int y, int r) {
        if (r < 0) throw new IllegalArgumentException("negative radius");
        this.x = x; this.y = y; this.r = r;
    }

    // 这个“副本构造方法”创建一个副本,可代替clone()方法
    public Circle(Circle original) {
        x = original.x;   // 直接从源对象中复制字段的值
        y = original.y;
        r = original.r;
    }

    // 公开的访问器方法,用于访问私有字段
    // 这也是为了封装数据
    public int getX() { return x; }
    public int getY() { return y; }
    public int getR() { return r; }

    // 返回对象的字符串表示形式
    @Override public String toString() {
        return String.format("center=(%d,%d); radius=%d", x, y, r);
    }

    // 测试与另一个对象是否相等
    @Override public boolean equals(Object o) {
        // 引用同一个对象?
        if (o == this) return true;
        // 类型不对,但不是null?
        if (!(o instanceof Circle)) return false;
        Circle that = (Circle) o;                // 校正为Circle类型
        if (this.x == that.x && this.y == that.y && this.r == that.r)
            return true;                         // 如果所有字段的值都相等
        else
            return false;                        // 如果字段的值不相等
    }

    // 有哈希码的对象才能在哈希表中使用
    // 相等的对象必须具有相等的哈希码
    // 不相等的对象可以具有相等的哈希码,但要尽量避免出现这种情况
    // 因为我们覆盖了equals()方法,所以必须覆盖这个方法
    @Override public int hashCode() {
        int result = 17;// 这个哈希码算法出自Joshua Bloch写的Effective Java
         result = 37*result + x;
        result = 37*result + y;
        result = 37*result + r;
        return result;
    }
```

```
    // 这个方法由Comparable接口定义
    // 比较这个Circle对象和另一个Circle对象
    // 如果这个对象小于另一个对象,返回负数
    // 如果这个对象等于另一个对象,返回零
    // 如果这个对象大于另一个对象,返回正数
    // Circle对象按照从上到下、从左到右排序,然后再比较半径的大小
    public int compareTo(Circle that) {
        // y坐标较大的圆较小
        long result = (long)that.y - this.y;
        // 如果y坐标相同,再比较x坐标
        if (result==0) result = (long)this.x - that.x;
        // 如果x坐标相同,再比较半径
        if (result==0) result = (long)this.r - that.r;

        // 相减时必须使用long类型的值,因为较大的正数和较小的负数
        // 之差可能会溢出int类型的取值范围。但是不能返回long类型的值
        // 因此返回表示符号的int类型值
        return Long.signum(result);
    }
}
```

5.2.1　toString()方法

toString() 方法的作用是返回对象的文本表示形式。连接字符串或使用 System.out.println() 等方法时，会自动在对象上调用这个方法。给对象提供文本表示形式，十分利于调试或记录日志，而且精心编写的 toString() 方法还能给报告生成等任务提供帮助。

Object 类中的 toString() 方法返回的字符串由对象所属的类名和对象的十六进制形式哈希码（由 hashCode() 方法计算得到，本章稍后会介绍）组成。这个默认的实现方式提供了对象的类型和标识两个基本信息，但一般并没什么用。示例 5-1 定义的 toString() 方法，返回一个人类可读的字符串，包含 Circle 类每个字段的值。

5.2.2　equals()方法

== 运算符测试两个引用是否指向同一个对象。如果要测试两个不同的对象是否相等，必须使用 equals() 方法。任何类都能覆盖 equals() 方法，定义专用的相等比较方式。Object.equals() 方法直接使用 == 运算符，只有两个对象是同一个对象时，才判定二者相等。

仅当两个不同的 Circle 对象的全部字段都相等时，示例 5-1 中定义的 equals() 方法才判定二者相等。注意，这个 equals() 方法先使用 == 运算符测试对象是否相同（一项优化措施），然后使用 instanceof 运算符检查另一个对象的类型，因为 Circle 对象只能和另一个 Circle 对象相等，而且 equals() 方法不能抛出 ClassCastException 异常。注意，instanceof 运算符还能排除 null：只要左侧操作数是 null，instanceof 运算符的计算结果就是 false。

5.2.3 hashCode()方法

只要覆盖了 `equals()` 方法，就必须覆盖 `hashCode()` 方法。`hashCode()` 方法返回一个整数，用于哈希表数据结构。如果两个对象经 `equals()` 方法测试是相等的，它们就要具有相同的哈希码。不相等的对象要具有不相等的哈希码（为了哈希表的操作效率），这一点很重要，但不是强制要求，最低要求是不相等的对象不能共用一个哈希码。为了满足最低要求，`hashCode()` 方法要使用稍微复杂的算法或位操作。

`Object.hashCode()` 方法和 `Object.equals()` 方法协同工作，返回对象的哈希码。这个哈希码基于对象的身份生成，而不是对象的相等性。（如果需要使用基于身份的哈希码，可以通过静态方法 `System.identityHashCode()` 获取 `Object.hashCode()` 方法的返回值。）

> 如果覆盖了 `equals()` 方法，必须覆盖 `hashCode()` 方法，这样才能保证相等的对象具有相同的哈希码。如果不这么做，程序可能会出现难以排查的问题。

在示例 5-1 中，因为 `equals()` 方法根据三个字段的值判断对象是否相等，所以 `hashCode()` 方法也基于这三个字段计算对象的哈希码。从 `hashCode()` 方法的代码中可以明确看出，如果两个 `Circle` 对象的字段值都相等，那么它们的哈希码也相等。

注意，示例 5-1 中的 `hashCode()` 方法没有直接相加三个字段的值，返回总和。这种实现方式算是合理，但还不够，因为如果两个圆的半径一样，但 *x* 和 *y* 坐标对调，哈希码依然相同。多次相乘和相加后，哈希码的值域会变大，因此能显著降低两个不相等的 `Circle` 对象具有相同哈希码的可能性。Joshua Bloch 写的 *Effective Java*（Addison Wesley 出版）一书对如何合理编写 `hashCode()` 方法提供了一个有用的攻略，和示例 5-1 中使用的类似。

5.2.4 Comparable::compareTo()方法

示例 5-1 包含一个 `compareTo()` 方法。这个方法由 `java.lang.Comparable` 接口而不是 `Object` 定义。但是这个方法经常要实现，所以也放在这一节介绍。`Comparable` 接口和其中的 `compareTo()` 方法用于比较类的实例，方式类似于比较数字的 `<`、`<=`、`>` 和 `>=` 运算符。如果一个类实现了 `Comparable` 接口，就可以比较一个实例是小于、大于还是等于另一个实例。这也表明，实现 `Comparable` 接口的类可以排序。

因为 `compareTo()` 方法不在 `Object` 类中声明，所以由每个类自行决定实例能否排序。如果能排序就定义 `compareTo()` 方法，实现实例排序的方式。示例 5-1 定义的排序方式把 `Circle` 对象当成一个页面中的单词，然后再比较。首先从上到下排序圆：*y* 坐标大的圆小于 *y* 坐标小的圆。如果两个圆的 *y* 坐标相同，再从左到右排序：*x* 坐标小的圆小于 *x* 坐标大的圆。如

果两个圆的 x 坐标和 y 坐标都相同，那就比较半径：半径小则圆也小。注意，按照这种排序方式，只有圆的三个字段都相等圆才相等。因此，compareTo() 方法定义的排序方式和 equals() 方法定义的相等条件是一致的。这么做非常合乎情理（但不是强制要求）。

compareTo() 方法返回一个 int 类型的值，这个值需要进一步说明。如果当前对象（this）小于传入的对象，compareTo() 方法应该返回一个负数；如果两个对象相等，应该返回 0；如果当前对象大于传入的对象，应该返回一个正数。

5.2.5　clone()方法

Object 类定义了一个名为 clone() 的方法，这个方法的作用是返回一个对象，并把这个对象的字段设为和当前对象一样。clone() 方法不常用，原因有两个。其一，只有类实现了 java.lang.Cloneable 接口，这个方法才有用。Cloneable 接口没有定义任何方法（是个标记接口），因此若想实现这个接口，只需在类签名的 implements 子句中列出这个接口即可。其二，clone() 方法声明为 protected，因此，如果想让其他类复制你的对象，你的类必须实现 Cloneable 接口，并覆盖 clone() 方法，而且要把 clone() 方法声明为 public。

示例 5-1 中的 Circle 类没有实现 Cloneable 接口，而是定义一个*副本构造方法*，用于创建 Circle 对象的副本：

```
Circle original = new Circle(1, 2, 3); // 普通构造方法
Circle copy = new Circle(original);    // 副本构造方法
```

clone() 方法很难正确实现，而副本构造方法实现起来更容易也更安全。若想让 Circle 类可克隆，要在 implements 子句中加入 Cloneable，然后在类的主体中添加下述方法：

```
@Override public Object clone() {
  try { return super.clone(); }
  catch(CloneNotSupportedException e) { throw new AssertionError(e); }
}
```

5.3　面向对象设计要略

本节介绍 Java 面向对象设计的几个相关技术，但不是很全面，只是为了展示一些示例。建议读者再阅读其他资料，例如前面提到的 Joshua Bloch 写的 *Effective Java* 一书。

本节先介绍 Java 定义常量时使用的良好实践，然后介绍使用 Java 的面向对象能力进行建模和领域对象设计的几种方式，最后介绍 Java 中一些常用设计模式的实现方式。

5.3.1　常量

前面说过，常量可以在接口中定义。实现某个接口的任何类都会继承这个接口中定义的常

量，而且使用起来就像是直接在类中定义的一样。重点是，这么做不需要在常量前加上接口的名称，也不需要以任何形式实现常量。

如果要在多个类中使用一组常量，更适合在一个接口中定义这些常量，需要使用这些常量的类实现这个接口即可。例如，客户端类和服务器类在实现网络协议时，就可以把细节（例如连接和监听的端口号）存储在一些符号常量中。举个实例，`java.io.ObjectStreamConstants` 接口。这个接口为对象序列化协议定义了一些常量，`ObjectInputStream` 和 `ObjectOutputStream` 类都实现了这个接口。

从接口中继承常量的主要好处是，能减少输入的代码量，因为无需指定定义常量的类型。但是，除了 `ObjectStreamConstants` 接口之外，并不推荐这么做。常量是实现细节，不该在类签名的 `implements` 子句中声明。

更好的方式是在类中定义常量，而且使用时要输入完整的类名和常量名。使用 `import static` 指令从定义常量的类中导入常量，可以减少输入的代码量。详情参见 2.10 节。

5.3.2　用接口还是抽象类

Java 8 的出现从根本上改变了 Java 的面向对象编程模型。在 Java 8 以前，接口纯粹是 API 规范，不包含实现。如果接口有大量实现，往往会导致代码重复。

为了解决这个问题，Java 设计者开发了一种代码模式。这个模式实现的基础是，抽象类无需完全抽象，可以包含部分实现，供子类使用。某些情况下，很多子类都可以沿用抽象超类提供的方法实现。

这种模式由两部分组成：一部分是一个接口，为基本方法制定 API 规范；另一部分是一个抽象类，初步实现这些方法。`java.util.List` 接口和与之匹配的 `java.util.AbstractList` 类是个很好的例子。JDK 提供的 `List` 接口的两个主要实现（`ArrayList` 和 `LinkedList`），都是 `AbstractList` 类的子类。下面再举个例子：

```
// 这是个简单的接口,表示可以放入一个矩形框中的形状
// 只要类想被当成RectangularShape类型,就可以从零开始实现这些方法
public interface RectangularShape {
    void setSize(double width, double height);
    void setPosition(double x, double y);
    void translate(double dx, double dy);
    double area();
    boolean isInside();
}

// 这是上述接口的部分实现
// 很多子类都可以以这些实现为基础
public abstract class AbstractRectangularShape
                        implements RectangularShape {
    // 形状的位置和尺寸
```

```
        protected double x, y, w, h;

        // 接口中部分方法的默认实现
        public void setSize(double width, double height) {
         w = width; h = height;
        }
        public void setPosition(double x, double y) {
         this.x = x; this.y = y;
        }
        public void translate (double dx, double dy) { x += dx; y += dy; }
    }
```

Java 8 引入的默认方法显著改变了这种情况。4.1.5 节说过，现在，接口可以包含实现代码。这意味着，如果定义的抽象类型（例如 Shape）可能有多个子类型（例如 Circle、Rectangle、Square），要面临一个抉择：用接口还是抽象类。因为接口和抽象类有很多类似的特性，所以有时并不确定应该使用哪个。

记住，如果一个类扩展了抽象类就不能再扩展其他类，而且接口依然不能包含任何非常量字段。也就是说，在 Java 中如何使用面向对象技术，仍有一些限制。

接口和抽象类之间的另一个重要区别和兼容性有关。如果你定义的一个接口是公开 API 的一部分，而后来想在接口中添加一个新的强制方法，那么已经实现这个接口的所有类都会出问题——也就是说，接口中添加的新方法必须声明为默认方法，并提供实现。但是，如果使用抽象类，可以放心添加非抽象方法，而不用修改已经扩展这个抽象类的类。

> 在这两种情况下，添加新方法都可能与子类中名称和签名相同的方法起冲突——此时，子类中的方法优先级更高。鉴于此，添加新方法时一定要谨慎，如果方法名对某个类型而言“显而易见”，或者方法可能有多个意义，尤其要小心。

一般来说，需要制定 API 规范时，推荐选择接口。接口中的强制方法不是默认方法，因为它们是 API 的一部分，实现方要提供有效的实现。当方法是真正可选的，或者只有一种可能的实现方式时，才应该使用默认方法。提供函数组合功能的 java.util.function.Function 接口是第二种情况的一例——函数只能使用一种标准方式组合，而且令人难以置信的是，只要方式合理就能覆盖默认的 compose() 方法。

最后，我要说一下，以前只注明接口中哪些方法是“可选的”，如果程序员不想实现这些方法就直接抛出 java.lang.UnsupportedOperationException 异常。这种做法问题多多，不要在新代码中使用。

5.3.3 实例方法还是类方法

实例方法是面向对象编程的关键特性之一，但并不是说应该避免使用类方法。很多情况

下，完全有理由定义类方法。

> 记住，在 Java 中，类方法使用关键字 static 声明，而且“静态方法”和“类方法”这两个术语指的是同一个概念。

例如，对 Circle 类而言，你可能经常要计算圆的面积。此时只需要半径，而不用创建一个 Circle 对象来表示这个圆。因此，使用类方法更便利：

```
public static double area(double r) { return PI * r * r; }
```

一个类完全可以定义多个同名方法，只要参数不同就行。上述 area() 方法是个类方法，因此没有表示 this 的隐式参数，但必须有一个参数用于指定圆的半径——就是这个参数把这个方法和同名实例方法区分开的。

下面再举个例子，说明应该使用实例方法还是类方法。假如我们要定义一个名为 bigger() 的方法，比较两个 Circle 对象，看哪一个半径较大。我们可以把 bigger() 定义为实例方法，如下所示：

```
// 比较隐式参数“this”表示的圆和显示参数“that”表示的圆
// 返回较大的那个圆
public Circle bigger(Circle that) {
  if (this.r > that.r) return this;
  else return that;
}
```

我们还可以把 bigger() 定义为类方法，如下所示：

```
// 比较圆a和b,返回半径较大的那个
public static Circle bigger(Circle a, Circle b) {
  if (a.r > b.r) return a;
  else return b;
}
```

如果有两个 Circle 对象，x 和 y，我们既可以使用实例方法，也可以使用类方法判断哪个圆较大。不过，调用这两个方法的句法有显著区别：

```
// 实例方法,也可以使用y.bigger(x)
Circle biggest = x.bigger(y);
Circle biggest = Circle.bigger(x, y); // 静态方法
```

两个方法都能很好地完成比较操作，而且从面向对象设计的角度来看，没有哪个方法“更正确”。从外观上看，实例方法更像是面向对象，但调用句法有点不对称。遇到这种情况时，使用实例方法还是类方法完全由设计方式而定。在实际情况中，应该有一种方式更自然。

关于System.out.println()

前面，我们多次遇到 System.out.println() 方法。这个方法的作用是把输出显示在终端窗口或控制台中。我们还没说明为什么这个方法的名称这么长、这么笨拙，也没有说明两个点号的作用。现在，你已经理解了类字段和实例字段，以及类方法和实例方法，那么再理解这个方法就容易了。System 是一个类，这个类有一个公开的类字段 out，这个字段的值是一个类型为 java.io.PrintStream 的对象，而这个对象有一个名为 println() 的实例方法。

我们可以使用静态导入指令 import static java.lang.System.out;，把这个方法的名称稍微变短一点儿，使用 out.println() 引用这个打印方法。不过，既然这是个实例方法，其名称还可以进一步缩短。

5.3.4 合成还是继承

面向对象设计时，继承不是唯一可选择的技术。对象可以包含其他对象的引用，因此，一个大型概念单元可以由多个小型组件组成——这种技术叫合成（composition）。与此有关的一个重要技术是委托（delegation）：某个特定类型的对象保存一个引用，指向一个兼容类型的附属对象，而且把所有操作都交给这个附属对象完成。这种技术一般使用接口类型实现，如下面的示例所示，这个示例构建软件公司的雇员架构模型：

```
public interface Employee {
  void work();
}

public class Programmer implements Employee {
  public void work() { /* 计算机编程 */ }
}

public class Manager implements Employee {
  private Employee report;

  public Manager(Employee staff) {
    report = staff;
  }

  public Employee setReport(Employee staff) {
    report = staff;
  }

  public void work() {
    report.work();
  }
}
```

在这个示例中，Manager 类把 work() 操作委托给直接下属完成，Manager 对象没有做任何实际工作。这种模式有些变体，发出委托的类完成一些工作，委托对象只完成部分工作。

另一个有用的相关技术是修饰模式（decorator pattern）。这种模式提供了扩展对象功能

的能力，在运行时也能扩展，但设计时要稍微付出一些额外劳动。下面举个例子说明修饰模式。这个例子为快餐店出售的墨西哥卷饼建模，简单起见，只修饰卷饼的一个属性——价格：

```
// 墨西哥卷饼的基本接口
interface Burrito {
  double getPrice();
}

// 具体实现——标准尺寸卷饼
public class StandardBurrito implements Burrito {
  private static final double BASE_PRICE = 5.99;

  public double getPrice() {
    return BASE_PRICE;
  }
}

// 超大尺寸卷饼
public class SuperBurrito implements Burrito {
  private static final double BASE_PRICE = 6.99;

  public double getPrice() {
    return BASE_PRICE;
  }
}
```

这个例子涵盖了在售的墨西哥卷饼——两种不同尺寸、不同价格的卷饼。下面我们来增强这个例子，提供两种可选的配料——墨西哥辣椒和鳄梨酱。设计的关键是使用一个抽象类，让这两个可选的配料扩展：

```
/*
 * 这个类是Burrito接口的修饰器
 * 表示墨西哥卷饼可选的配料
 */
public abstract class BurritoOptionalExtra implements Burrito {
    private final Burrito burrito;
    private final double price;

    // 这个构造方法声明为protected,目的是保护默认构造方法
    // 以及避免劣质的客户端代码直接实例化这个基类
    protected BurritoOptionalExtra(Burrito toDecorate,
          double myPrice) {
        burrito = toDecorate;
        price = myPrice;
    }

    public final double getPrice() {
        return (burrito.getPrice() + price);
    }
}
```

把 BurritoOptionalExtra 类声明为 abstract，并把构造方法声明为 protected，这样只有创建子类的实例才能获得有效的 BurritoOptionalExtra 对象，因为子类提供了公开的构造方法（这样也能避免客户端代码设定配料的价格）。

下面测试一下上述实现方式：

```
Burrito lunch = new Jalapeno(new Guacamole(new SuperBurrito()));
// 这个墨西哥卷饼的总价应该是$8.09
System.out.println("Lunch cost: "+ lunch.getPrice());
```

修饰模式使用非常广泛，不仅局限于 JDK 中的实用类。第 10 章介绍 Java I/O 时会见到更多使用修饰器的示例。

5.3.5 字段继承和访问器

Java 为设计状态的继承时可能遇到的问题提供了多种解决方案。程序员可以选择用 protected 修饰字段，允许子类直接访问这些字段（也可以设定字段的值）。或者，可以提供访问器方法，直接读取对象的字段（如果需要，也可以设定字段的值），这么做仍能有效封装数据，而且可以把字段声明为 private。

我们再看一下第 3 章末尾举的 PlaneCircle 示例，这里明确展示了字段继承：

```
public class Circle {
  // 这是通用的常量,所以要保证声明为public
  public static final double PI = 3.14159;

  protected double r;     // 通过受保护字段继承的状态

  // 限制半径取值的方法
  protected void checkRadius(double radius) {
    if (radius < 0.0)
      throw new IllegalArgumentException("radius may not < 0");
  }

  // 非默认的构造方法
  public Circle(double r) {
    checkRadius(r);
    this.r = r;
  }

  // 公开的数据访问器方法
  public double getRadius() { return r; }
  public void setRadius(double r) {
    checkRadius(r);
    this.r = r;
  }

  // 操作实例字段的方法
```

```
  public double area() { return PI * r * r; }
  public double circumference() { return 2 * PI * r; }
}

public class PlaneCircle extends Circle {
  // 自动继承了Circle类的字段和方法
  // 因此只要在这里编写新代码
  // 新实例字段,存储圆心的位置
  private final double cx, cy;

  // 新构造方法,用于初始化新字段
  // 使用特殊的句法调用构造方法Circle()
  public PlaneCircle(double r, double x, double y) {
    super(r);      // 调用超类的构造方法Circle()
    this.cx = x;   // 初始化实例字段cx
    this.cy = y;   // 初始化实例字段cy
  }

  public double getCentreX() {
    return cx;
  }

  public double getCentreY() {
    return cy;
  }

  // area()和circumference()方法继承自Circle类
  // 新实例方法,检查点是否在圆内
  // 注意,这个方法使用了继承的实例字段r
  public boolean isInside(double x, double y) {
    double dx = x - cx, dy = y - cy;
    // 勾股定理
    double distance = Math.sqrt(dx*dx + dy*dy);
    return (distance < r);                   // 返回true或false
  }
}
```

除了上述方式之外，还可以使用访问器方法重写 PlaneCircle 类，如下所示：

```
public class PlaneCircle extends Circle {
  // 其他代码和前面一样
  // 超类Circle的r字段可以声明为private,因为现在不直接访问r字段了

  // 注意,现在使用访问器方法getRadius()
  public boolean isInside(double x, double y) {
    double dx = x - cx, dy = y - cy;            // 到圆心的距离
    double distance = Math.sqrt(dx*dx + dy*dy); // 勾股定理
    return (distance < getRadius());
  }
}
```

上述两种方式都是合法的 Java 代码，但有一些区别。3.5 节说过，在类外部可写的字段，一般不是建模对象状态的正确方式。在 5.5 节和 6.5 节会看到，这么做其实会对程序的运

行状态造成不可恢复的损坏。

糟糕的是，Java 中的 protected 关键字允许子类和同一个包中的类访问字段（和方法）。加之任何人都能把自己编写的类放入任何指定的包（不含系统包），这就意味着，在 Java 中使用继承的受保护状态有潜在缺陷。

Java 没有提供只能在声明成员的类和子类中访问成员的机制。

鉴于上述原因，在子类中一般最好使用（公开的或受保护的）访问器方法访问状态——除非把继承的状态声明为 final，才完全可以使用继承的受保护状态。

5.3.6　单例

单例模式（singleton pattern）是人们熟知的另一个设计模式，用来解决只需要为类创建一个实例这种设计问题。Java 为单例模式提供了多种实现方式。这里我们要使用一种稍微复杂的方式，但有个好处，它能十分明确地表明为了安全实现单例模式要做些什么：

```
public class Singleton {
  private final static Singleton instance = new Singleton();
  private static boolean initialized = false;

  // 构造方法
  private Singleton() {
    super();
  }

  private void init() {
    /* 做初始化操作 */
  }

  // 这个方法是获取实例引用的唯一方式
  public static synchronized Singleton getInstance() {
    if (initialized) return instance;
    instance.init();
    initialized = true;
    return instance;
  }
}
```

为了有效实现单例模式，重点是要确保不能有超过一种创建实例的方式，而且要保证不能获取处于未初始化状态的对象引用（本章稍后会详细说明这一点）。为此，我们需要一个声明为 private 的构造方法，而且只调用一次。在这个 Singleton 类中，我们只在初始化私有静态变量 instance 时才调用构造方法。而且，我们还把创建唯一一个 Singleton 对象

的操作和初始化操作分开，把初始化操作放入私有方法 init() 中。

使用这种机制，获取 Singleton 的唯一实例只有一种方式——通过静态辅助方法 getInstance()。getInstance() 方法检查标志 initialized，确认对象是否已经处于激活状态。如果没有激活，getInstance() 方法调用 init() 方法激活对象，然后把 initialized 设为 true，所以下次请求创建 Singleton 的实例时，不会再做进一步初始化操作。

最后还要注意，getInstance() 方法使用 synchronized 修饰。第 6 章会详细说明这么做的意义和原因。现在，你只需知道，加上 synchronized 是为了防止在多线程程序中使用 Singleton 时得到意外的结果。

单例虽然是最简单的模式之一，但经常过度使用。如果使用得当，单例是很有用的技术，但如果一个程序中有太多单例类，往往表明代码设计得不好。

单例模式有一些弊端：难测试，难与其他类分开。而且，在多线程代码中使用时需要小心。虽然如此，单例模式仍然很重要，开发者要熟练掌握，别不小心重新发明轮子。单例模式一般用于管理配置，但是现代的代码经常使用自动为程序员提供单例的框架（一般是*依赖注入*），而不是自己动手编写 Singleton 类（或类似的类）。

5.4 异常和异常处理

2.6.3 节介绍过已检异常和未检异常。本节进一步讨论异常在设计方面的问题，以及如何在你自己的代码中使用异常。

记住，在 Java 中，异常是对象。这个对象的类型是 java.lang.Throwable，更准确地说是 Throwable 类的子类，更具体地描述发生的异常是什么类型。Throwable 类有两个标准子类：java.lang.Error 和 java.lang.Exception。Error 类的子类对应的异常表示不可恢复的问题，例如，虚拟机耗尽了内存，或类文件损坏了，无法读取。这种异常可以捕获并处理，但很少这么做——这种异常就是前面提到的未检异常。

而 Exception 类的子类对应的异常表示没那么严重的状况，可以捕获并处理，例如：java.io.EOFException，表示到达文件的末尾；java.lang.ArrayIndexOutOfBoundsException，表示程序尝试读取的元素超出了数组的末端。这种异常是第 2 章介绍过的已检异常（RuntimeException 的子类是个例外，仍然属于未检异常）。本书使用“异常”这个术语指代所有异常对象，不管是 Exception 类型还是 Error 类型。

因为异常是对象，所以可以包含数据，而且异常所属的类可以定义方法，操作这些数据。

Throwable 类及其所有子类都包含一个 String 类型的字段，存储一个人类可读的错误消息，描述发生的异常状况。这个字段的值在创建异常对象时设定，可以使用 getMessage() 方法从异常对象中读取。多数异常都只包含这个消息，但少数异常还包含其他数据。例如，java.io.InterruptedIOException 异常包含一个名为 bytesTransferred 的字段，表示在异常状况中断传输之前完成了多少输入或输出。

自己设计异常时，要考虑建模异常对象需要哪些额外信息。这些信息一般是针对中断的操作和遇到的异常状况（例如前面的 java.io.InterruptedIOException 异常）。

在应用设计中使用异常时要做些权衡。使用已检异常的话，意味着编译器能处理（或顺着调用堆栈向上冒泡）可能恢复或重试的已知状况，还意味着更难忘记处理错误，因此能减少由于忘记处理错误状况而导致系统在生产环境中崩溃的几率。

另一方面，就算理论上某些状况建模为已检异常，有些应用也无法从这些状况中恢复。例如，如果一个应用需要读取在文件系统特定位置存储的配置文件，而应用启动时找不到这个文件，尽管 java.io.FileNotFoundException 是已检异常，但除了打印错误消息并退出之外，这个应用别无他法。遇到这种情况时，假若强制处理或冒泡无法恢复的异常，近乎于背道而驰。

设计异常机制时，应该遵循下述良好的做法：

- 考虑要在异常中存储什么额外状态——记住，异常也是对象；
- Exception 类有四个公开的构造方法，一般情况下，自定义异常类时这四个构造方法都要实现，可用于初始化额外的状态，或者定制异常消息；
- 不要在你的 API 中自定义很多细致的异常类——Java I/O 和反射 API 都因为这么做了而受人诟病，所以别让使用这些包时的情况变得更糟；
- 别在一个异常类型中描述太多状况——例如，实现 JavaScript 的 Nashorn 引擎（Java 8 的新功能）一开始有超多粗制滥造的异常，不过在发布之前修正了。

最后，还要避免使用两种处理异常的反模式：

```
// 不要捕获异常而不处理
try {
  someMethodThatMightThrow();
} catch(Exception e){
}

// 不要捕获,记录日志后再重新抛出异常
try {
  someMethodThatMightThrow();
} catch(SpecificException e){
  log(e);
  throw e;
}
```

第一个反模式直接忽略近乎一定需要处理的异常状况（甚至没有在日志中记录）。这么做会增大系统其他地方出现问题的可能性——出现问题的地方可能会离原来的位置很远。

第二个反模式只会增加干扰——虽然记录了错误消息，但没真正处理发生的问题——在系统高层的某部分代码中还是要处理这个问题。

5.5　Java编程的安全性

有些编程语言被称为*类型安全*语言，但程序员使用这个术语时要表达的意思却很宽松。“类型安全”有很多解读和定义方式，而且各种方式之间并不都有关联。对我们要讨论的话题来说，类型安全最适合理解为编程语言的一个属性，其作用是避免运行时把数据识别为错误的类型。类型安全与否是相对的，正确的理解方式应该是，一门语言比（或没有）另一门语言安全，而不能直接断定一门语言是绝对安全的或绝对不安全的。

Java 的类型系统是静态的，能避免很多问题，例如，如果程序员试图把不兼容的值赋值给变量，会导致编译出错。但是，Java 的类型安全并不完美，因为任何两种引用类型之间都可以通过校正相互转换——如果值之间不兼容，这种转换在运行时会失败，抛出 `ClassCastException` 异常。

本书所说的安全性和更宽泛的正确性分不开，也就是说，我们要站在程序的角度，而不是语言的角度来探讨安全性。这强调了一个问题，即代码的安全不是由任何一门广泛使用的语言决定的，而要由程序员付出足够的努力（并严格遵守编程准则），确保写出的代码真正安全且正确。

为了得到安全的程序，我们要使用图 5-1 表示的抽象状态模型。安全的程序具有以下特征：

- 所有对象在创建后都处于一种合法状态；
- 外部可访问的方法在合法状态之间转换对象；
- 外部可访问的方法绝对不能返回状态不一致的对象；
- 弃用对象之前，外部可访问的方法必须把对象还原为合法状态。

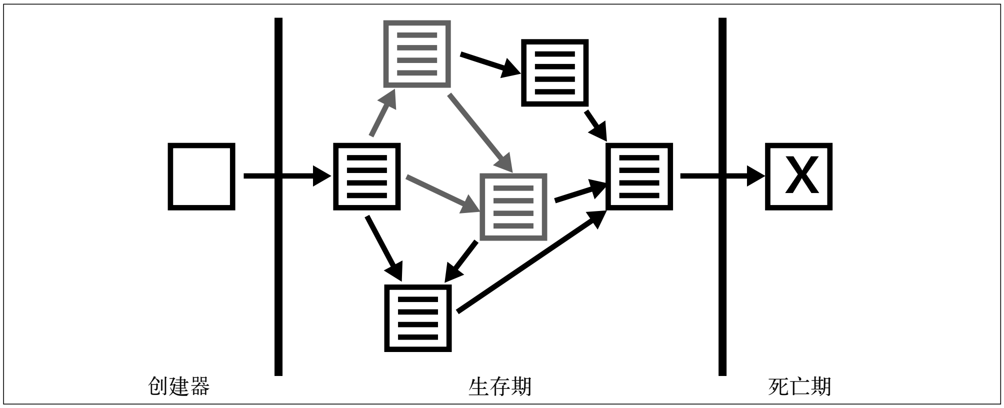


图 5-1：程序的状态转换

其中，“外部可访问”的方法是指声明为 `public`、`protected` 或者对包私有的方法。上述特征为安全的程序定义了一个合理的模型，而且按照这种方式定义的抽象类型，它的方法能保证状态的一致性。满足上述条件的程序就可以称为“安全的程序”，而不用管程序使用何种语言实现。



私有方法不用保证对象在使用前后都处于合法状态，因为私有方法不能在外部代码中调用。

你可能想到了，如果想在大量代码中让状态模型和方法都满足上述特征，需要付出相当多的精力。对 Java 等语言来说，因为程序员能直接创建由多个线程执行的抢占式多任务程序，所以要付出的精力更多。

介绍完 Java 的面向对象设计要略之后，关于 Java 语言和平台还有一个方面需要扎实地理解——内存管理和并发编程。这是 Java 平台最复杂的知识，但掌握之后会获益良多。这是下一章要讨论的话题，介绍完之后本书第一部分也就结束了。

第 6 章

Java实现内存管理和并发编程的方式

本章介绍 Java 平台处理并发（多线程）编程和内存管理的方式。这两个话题有内在联系，所以放在一起介绍。本章包含以下内容：

- 介绍 Java 管理内存的方式；
- 标记清除垃圾回收（Garbage Collection，GC）算法基础；
- HotSpot JVM 根据对象的生命周期优化垃圾回收的方式；
- Java 的并发基元；
- 数据的可见性和可变性。

6.1　Java内存管理的基本概念

在 Java 中，对象占用的内存在不需要使用对象时会自动回收。这个过程叫作*垃圾回收*（或自动内存管理）。垃圾回收这项技术在 Lisp 等语言中已经存在好多年了，习惯使用 C 和 C++ 等语言的程序员要花点儿时间适应，因为在这些语言中必须调用 `free()` 函数或使用 `delete` 运算符才能回收内存。

> Java 是一门用起来很舒心的语言，原因之一是，不用动手销毁自己创建的每一个对象。也是基于这个原因，较之不支持自动垃圾回收机制的语言，使用 Java 编写的程序较少存在缺陷。

不同的虚拟机使用不同的方式实现垃圾回收，而且规范没有对如何实现垃圾回收做强制要

求。本章后面会讨论 HotSpot JVM，这虽然不是你会用到的唯一一个 JVM，但部署在服务器端的应用最常使用这个 JVM，而且 HotSpot 是现代化生产环境使用的典型 JVM。

6.1.1 Java中的内存泄露

Java 支持垃圾回收，因此可以显著减少内存泄露的发生几率。分配的内存没有回收，就会发生内存泄露。乍看起来，垃圾回收似乎能避免一切内存泄露的发生，因为这个机制能回收所有不再使用的对象。

但是，在 Java 中，如果不再使用的对象存在有效（但不再使用）的引用，仍然会发生内存泄露。例如，如果某个方法运行的时间很长（或者一直运行下去），那么这个方法中的局部变量会一直保存对象的引用，远超实际所需的时间，如下述代码所示：

```
public static void main(String args[]) {
  int bigArray[] = new int[100000];

  // 对bigArray做些计算,得到一个结果
  int result = compute(bigArray);

  // 不再需要使用bigArray了。如果没有引用指向bigArray,就会被垃圾回收
  // 但是bigArray是局部变量,在方法返回之前始终指向那个数组
  // 可是这个方法还没有返回,因此我们要自己动手销毁引用
  // 告知垃圾回收程序回收这个数组
  bigArray = null;

  // 无限循环,处理用户的输入
  for(;;) handle_input(result);
}
```

使用 `HashMap` 或类似的数据结构关联两个对象时，也可能会发生内存泄露。就算有一个对象不再需要使用了，哈希表中仍然存有两个对象之间的关联，因此在回收哈希表之前，这两个对象一直存在。如果哈希表的生命周期比其中的对象长得多，就可能导致内存泄露。

6.1.2 标记清除算法简介

JVM 确切知道它分配了哪些对象和数组，这些对象和数组存储在某种内部数据结构中，我们称这种数据结构为*分配表*（allocation table）。JVM 还能区分每个栈帧（stack frame）里的局部变量指向堆（heap）里的哪个对象或数组。最后，JVM 能追踪堆中对象和数组保存的引用，不管引用多么迂回，都能找到所有仍然被引用的对象和数组。

因此，运行时能判断已经分配内存的对象什么时候不再被其他活动对象或变量引用。遇到这种对象时，解释器知道它可以放心地回收这个对象的内存，然后回收内存。注意，垃圾回收程序还能检测到相互引用的对象，如果没有其他活动对象引用这些对象，就将其内存回收。

在应用线程的堆栈跟踪中，从其中一个方法的某个局部变量开始，沿着引用链，如果最终能找到一个对象，我们称这个对象为*可达对象*（reachable object）。这种对象也叫*活性对象*。[1]

> 除了局部变量之外，引用链还可以从其他几个地方开始。通向可达对象的引用链根部一般称为 GC Root。

知道这些简单的定义之后，我们来看一种基于这些原则回收垃圾的简单方式。

6.1.3　基本标记清除算法

垃圾回收过程经常使用（也是最简单）的算法是*标记清除*（mark and sweep）。整个过程分为三步。

(1) 迭代分配表，把每个对象都标记为“已死亡”。

(2) 从指向堆的局部变量开始，顺着遇到的每个对象的全部引用向下，每遇到一个之前没见过的对象或数组，就把它标记为“存活”。像这样一直向下，直到找出能从局部变量到达的所有引用为止。

(3) 再次迭代分配表，回收所有没标记为“存活”的对象在堆中占用的内存，然后把这些内存放回可用内存列表中，最后把这些对象从分配表中删除。

> 上面概述的标记清除过程是这个算法理论上最简单的形式。在后面的几节中会看到，真正的垃圾回收程序做的事情比这要多。上面的概述是为了打好理论基础，目的就是易于理解。

因为所有对象的内存都由分配表分配，所以用完堆内存之前会触发垃圾回收程序。在上述对标记清除算法的描述中，垃圾回收程序需要互斥存取整个堆，因为应用代码一直在运行中，会不断创建和修改对象，导致结果腐化。

图 6-1 展示了在应用线程运行过程中尝试回收对象的后果。

注 1：从 GC Roots 对象开始向下穷根揭底的探索过程称为活性对象的*传递闭包*（transitive closure）——这个术语从图论抽象数学中借用而来。

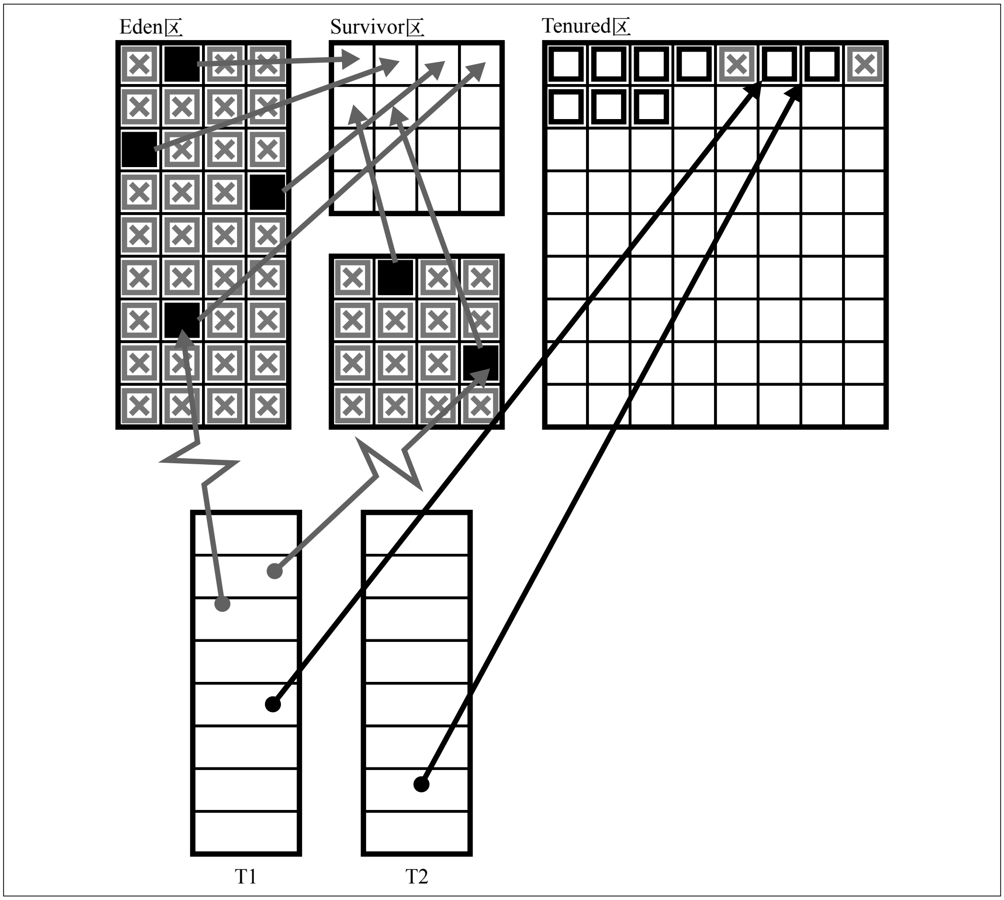


图 6-1：堆内存的变化

为了避免发生这种问题，在上述简单的垃圾回收过程中，应用线程会停顿一下（这个停顿叫 Stop-The-World，STW）——先停止所有应用线程，然后回收垃圾，最后继续运行应用线程。应用线程执行到一个*安全点*（safepoint）时，例如循环的开始处或即将调用方法时，运行时会让应用线程停顿一下，因为运行时知道在安全点可以放心地停止运行应用线程。

开发者有时会担心这种停顿，但是对大多数主流应用场景来说，Java 都运行在操作系统之上，进程会不断交替进出处理器内核，因此一般无需担心这些短暂的额外停顿。HotSpot 会做大量工作来优化垃圾回收，减少 STW 时间，这一点对减轻应用的工作负担来说十分重要。下一节会介绍一些优化措施。

6.2 JVM优化垃圾回收的方式

对第 1 章介绍的软件来说，运行时会对其做些处理，其中一个很好的例子是*弱代假设*

（Weak Generational Hypothesis，WGH）。简单来说，在这个假设中，对象常常处于少数几个预期生命周期之一（这些预期生命周期叫“代”）。

一般来说，对象的生命期非常短（有时把这种对象叫*瞬时对象*），不久就会当作垃圾回收。然而，有些少量对象会存在得久一点，因此注定会成为程序长期状态的一部分（程序的长期状态有时称为程序的*工作集*）。这种现象可通过图 6-2 表示，这幅图绘制的是预期生命周期中内存用量的变化。

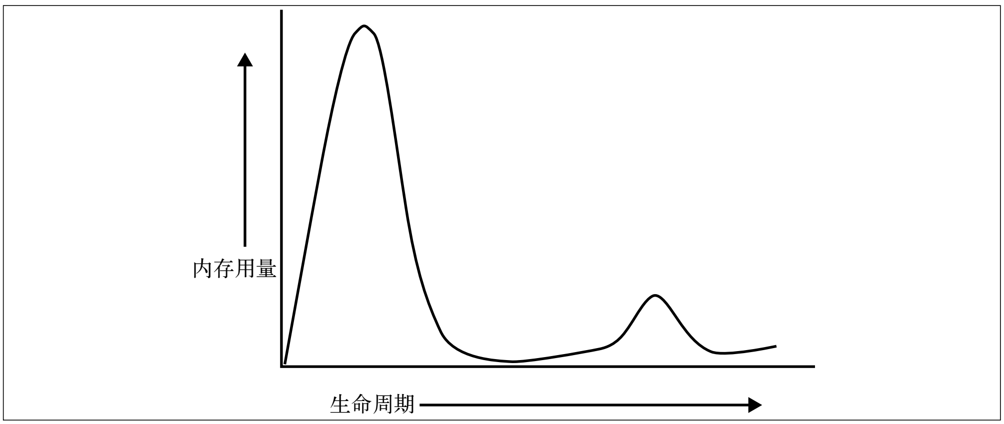


图 6-2：弱代假设

这种变化趋势不是由静态分析推知的，在监测软件运行时行为时可以看出，在不同的应用场景中，这明显都是事实。

HotSpot JVM 有一个垃圾回收子系统，专门利用弱代假设。本节，我们要讨论如何在生命期短的对象（大多数情况下对象的寿命都短）上使用这些技术。这些论述默认针对 HotSpot，不过其他服务器端使用的 JVM 一般也都使用类似或相关的技术。

最简单的*分代垃圾回收程序*只会留意弱代假设，因为分代垃圾回收程序认为，较之充分使用弱代假设，做额外的簿记来监控内存，投入少收效大。在最简单的分代垃圾回收程序中，往往只有两代，这两代一般称为新生代和老年代。

筛选回收

在上述标记清理算法的清理阶段，一个一个回收对象，然后把各个对象占用的空间放回可用内存列表。然而，如果弱代假设成立，而且在任何一个垃圾回收循环中大多数对象都已“死亡”，那么使用另一种方式回收空间似乎更合理。

新的回收方式把堆内存分成多个独立的内存空间，每次回收垃圾时，只为活性对象分配空间，并把这些对象移到另一个内存空间。这个过程叫作*筛选回收*（evacuation），执行这个

过程的回收程序叫作*筛选回收程序*。这种回收程序回收完毕后会清理整个内存空间，供以后重复使用。图 6-3 展示了筛选回收程序的工作方式。

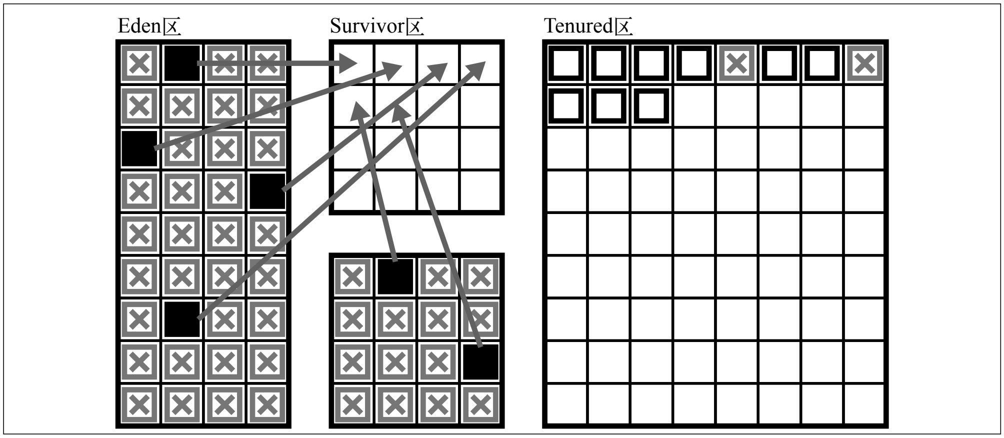


图 6-3：筛选回收程序

这种回收方式可能比上述简单的回收方式高效得多，因为根本不用管已死对象。垃圾回收循环执行的时间长短和活性对象的数量成正比，而不是已分配空间的对象数量。这种回收方式唯一的缺点是，要稍微花点儿时间簿记，因为要复制活性对象，但这点儿时间与获得的巨大收益相比不值一提。

> HotSpot 完全在用户空间中自行管理 JVM 堆，而且分配内存和释放内存时不用执行系统调用。对象一开始在 Eden 区（或叫 Nursery 区）创建，大多数生产环境使用的 JVM（至少 SE/EE 使用的 JVM）都会使用筛选回收策略回收 Eden 区的垃圾。

使用筛选回收程序的话，每个线程都可以单独分配内存。也就是说，每个应用线程都有一块连续的内存（叫*线程私有的分配缓冲区*），专门供这个线程分配新对象。为新对象分配内存时，只需把指针指向分配缓冲区，非常省事。

如果对象在回收操作即将开始之前创建，那么这个对象没有时间完成使命，在垃圾回收循环开始前就会“死亡”。在只有两代的回收程序中，这种生命期短的对象会被移入长存区，几乎相当于宣布“死刑”，然后等待下次回收循环将其回收。因为这种情况很少见（往往也很费事），执行上述操作似乎相当浪费资源。

为了缓和这种浪费，HotSpot 引入了 Survivor 区。Survivor 区用于保存前一次回收新生对象后存活下来的对象。筛选回收程序会在多个 Survivor 区之间来回复制存活下来的对象，直到超过*保有阈值*后，再把这些对象推给老年代。

详述 Survivor 区和调校垃圾回收程序的方式已经超出本书范畴。如果要把应用部署到生产环境，应该参阅专门的资料。

6.3 HotSpot堆

HotSpot JVM 的代码相当复杂，由一个解释器、一个即时编译器和一个用户空间内存管理子系统组成。HotSpot JVM 的代码使用 C 和 C++ 编写，还有相当多针对特定平台的汇编代码。

现在，我们来总结一下什么是 HotSpot 堆，再回顾一下它的基本特性。Java 堆是一块连续的内存，在启动 JVM 时创建，但一开始只会把部分堆分配给各个内存池。在应用运行的过程中，内存池会按需扩容。扩容由垃圾回收子系统完成。

> **堆中的对象**
>
> 应用线程在 Eden 区创建对象，不确定性垃圾回收循环会移除这些对象。这个垃圾回收循环在需要时（即内存不够用时）才会运行。堆分为两代：新生代和老年代。新生代由三个区组成：Eden 区和两个 Survivor 区；而老年代只有一个内存空间。
>
> 多次垃圾回收循环后存活下来的对象，最终会推给老年代。只回收新生代的回收操作消耗（所需计算）往往不大。HotSpot 使用的标记清除算法比目前为止我们见到的要高级，而且还会做额外的簿记，提升垃圾回收的性能。下一节介绍老年代，以及 HotSpot 如何处理生命期较长的对象。

6.3.1 回收老年代

讨论垃圾回收程序时，开发者还要知道两个重要的术语。

- *并行回收程序*

  使用多个线程执行回收操作的垃圾回收程序。

- *并发回收程序*

  可以和应用线程同时运行的垃圾回收程序。

到目前为止，我们见到的回收程序都是并行回收程序，而不是并发回收程序。默认情况下，老年代使用的回收程序也是并行标记清除回收程序（而不是并发回收程序），但是，HotSpot 允许植入不同的回收程序。例如，稍后在本节会见到 HotSpot 内置的 CMS 回收程序，这是并行回收程序，但基本上也是并发回收程序。

乍看起来，老年代默认使用的回收程序和新生代使用的回收程序类似，但二者有个重要的区别：老年代默认使用的回收程序不是筛选回收程序。回收老年代时，回收程序会整理老年代。这一点很重要，这样内存空间在使用的过程中不会产生碎片。

6.3.2 其他回收程序

这一节完全针对 HotSpot，但不会深入介绍，因为超出本书范畴了。不过你要知道，还有其他一些回收程序存在。如果不使用 HotSpot，你应该阅读 JVM 的文档，看看有什么其他选择。

1. 并发标记清除

HotSpot 中最常使用的替代回收程序是并发标记清除（Concurrent Mark and Sweep，CMS）回收程序。这个回收程序只能用来回收老年代，与一个回收新生代的并行回收程序配合使用。

> CMS 只适用于需要短暂停顿的应用，这些应用的停顿时间不能超过 STW 的几毫秒。这类应用极少，除了金融贸易类应用之外，很少有应用真正需要这么短的停顿时间。

CMS 是个很复杂的回收程序，往往很难有效调校。CMS 是个非常有用的工具，但部署时不能掉以轻心。CMS 有一些基本特性（如下所示）你要知道，但详细说明已经超出本书范畴。有兴趣的读者可以阅读专门的博客和邮件列表（例如，“Friends of jClarity”邮件列表经常讨论 GC 性能方面的问题）。

- CMS 只能回收老年代；
- 在多数 GC 循环中，CMS 都和应用线程一起运行，以便减少停顿时间；
- 应用线程不会像之前那样停顿很久；
- 分为六个阶段，都是为了缩减 STW 停顿时间；
- 把一次 STW 长停顿变成两次（往往很短的）STW 停顿；
- 簿记工作更多，CPU 时间也更长；
- 总体来说，GC 循环的时间更长；
- 默认情况下，并发运行时，GC 使用一半 CPU；
- 除了需要短暂停顿的应用之外，不要使用 CMS；
- 绝对不能在吞吐量大的应用中使用；
- 不会整理内存，如果内存碎片很多，会回滚到默认的（并行）回收程序。

2. G1

Garbage First 回收程序（简称 G1）是一个新的垃圾回收程序，在 Java 7 时代开发（Java 6 时代完成了部分准备工作）。G1 是一种短暂停顿回收程序，目的是取代 CMS。而且 G1 允许用户设定*停顿指标*，指定回收垃圾时停顿多久，以及多久停顿一次。和 CMS 不同的是，G1 适用于吞吐量较高的应用场合。

G1 使用粗粒度方式管理内存，把内存分成多个区，集中精力管理几乎充满垃圾的区，因

为这些区释放的内存最多。G1 是一种筛选回收程序，筛选各区时，会不断整理内存。

新开发一个通用的生产级回收程序，不是个简单的过程。因此，虽然 G1 已经开发了数年，但在 2014 年年初，多数评测仍显示 G1 没有 CMS 效率高。话虽如此，不过二者之间的差距在稳步减小，而且在某些应用场合中，G1 已经处于领先地位。在未来数月或数年中，G1 完全可能会变成最常用的短暂停顿回收程序。

最后，HotSpot 还有一个 Serial 回收程序（和 SerialOld 回收程序），以及一个“增量式 CMS”回收程序。这些回收程序都废弃了，不要再使用。

6.4　终结机制

有一种古老的资源管理技术叫*终结*（finalization），开发者应该知道有这么一种技术。然而，这种技术几乎完全废弃了，任何情况下，大多数 Java 开发者都不应该直接使用。

> 只有少数应用场景适合使用终结，而且只有少数 Java 开发者会遇到这种场景。如果有任何疑问，就不要使用终结，处理资源的 `try` 语句往往是正确的替代品。

终结机制的作用是自动释放不再使用的资源。垃圾回收自动释放的是对象使用的内存资源，不过对象可能会保存其他类型的资源，例如打开的文件和网络连接。垃圾回收程序不会为你释放这些额外的资源，因此，终结机制的作用是让开发者执行清理任务，例如关闭文件、中断网络连接、删除临时文件，等等。

终结机制的工作方式是这样的：如果对象有 `finalize()` 方法（一般叫作*终结方法*），那么不再使用这个对象（或对象不可达）后的某个时间会调用这个方法，但要在垃圾回收程序回收分配给这个对象的空间之前调用。终结方法用于清理对象使用的资源。

在 Oracle/OpenJDK 中，按照下述方式使用这种技术。

(1) 如果可终结的对象不可达了，会在内部*终结队列*中放一个引用，指向这个对象；而且，为了回收垃圾，这个对象会被标记为“存活”。

(2) 对象一个接着一个从终结队列中移除，然后调用各自的 `finalize()` 方法。

(3) 调用终结方法后，不会立即释放对象，因为终结方法可能会把 `this` 引用存储在某个地方（例如在某个类的公开静态字段中），让对象再次拥有引用，复活对象。

(4) 因此，调用 `finalize()` 方法后，垃圾回收子系统在回收对象之前，必须重新判断对象是否可达。

(5) 不过，就算对象复活了，也不会再次调用终结方法。

(6) 综上所述，定义了 `finalize()` 方法的对象一般（至少）会多存活一个 GC 循环（如果是生命期长的对象，会再多存活一个完整的 GC 循环）。

终结机制的主要问题是，Java 不确定什么时候回收垃圾，或者以什么顺序回收对象。因此，Java 平台无法确认什么时候（甚至是否）调用终结方法，或者以什么顺序调用终结方法。

因此，作为一种防止资源（例如文件句柄）稀少的自动清理机制，其设计是有缺陷的，因为不能保证终结机制运行得足够快，避免耗尽资源。

终结方法唯一真正有用的场景是，在一个类中使用本地方法，打开某个非 Java 资源。就算遇到这种情况，也更适合使用处理资源的块状 `try` 语句，但也可以声明一个 `public native finalize()` 方法（`close()` 方法会调用这个方法）——这个方法可以释放本地资源，包括不受 Java 垃圾回收程序控制的堆外内存。

终结机制的细节

为了少数适合使用终结机制的场景，下面列出一些额外细节，以及使用过程中的注意事项。

- 在没有回收全部重要的对象之前，JVM 可能就会退出，所以根本不会调用某些终结方法。遇到这种情况，操作系统会关闭网络连接等资源，并将其回收。然而，要注意，如果要删除文件的终结方法没有运行，操作系统不会删除那个文件。
- 为了确保在虚拟机退出前执行某些操作，Java 提供了 `Runtime::addShutdownHook` 钩子，在 JVM 退出前安全执行任意代码。
- `finalize()` 方法是实例方法，作用在实例上。没有等效的机制用来终结类。
- 终结方法是实例方法，没有参数，也不返回值。每个类只能有一个终结方法，而且必须命名为 `finalize()`。
- 终结方法可以抛出任何类型的异常或错误，但垃圾回收子系统自动调用终结方法时，终结方法抛出的任何异常或错误都会被忽略，这些异常或错误只会导致终结方法返回。

6.5 Java对并发编程的支持

线程的作用是提供一个轻量级执行单元——虽比进程小，但仍能执行任何 Java 代码。一般情况下，对操作系统来说，一个线程是一个完整的执行单元，但仍属于一个进程，进程的地址空间在组成该进程的所有线程之间共享。也就是说，每个线程都可以独立调度，而且有自己的栈和程序计数器，但会和同个进程中的其他线程共享内存和对象。

Java 平台从第一版开始就支持多线程编程，并向开发者开放了创建新线程的功能。创建新线程往往很简单，如下所示：

```
Thread t = new Thread(() -> {System.out.println("Hello Thread");});
t.start();
```

这段简短的代码创建并启动一个新线程，然后执行 lambda 表达式的主体，最后退出。如果你是使用过旧版 Java 的程序员，我告诉你，lambda 表达式其实会被转换成 `Runnable` 接口的实例，然后再传给 `Thread` 类的构造方法。

线程机制允许新线程和原有的应用线程以及 JVM 为了不同目的而创建的多个线程一起并发运行。

在大多数 Java 平台的实现中，应用线程都能访问操作系统调度程序控制的 CPU。调度程序是操作系统原生的一部分，用于管理处理器时间的时间片（也能禁止应用线程超出分配给它的时间）。

在最近几版 Java 中，越来越流行使用*运行时管理的并发*。因为基于很多原因，由开发者自行管理线程已经不能满足需求了。而运行时应该提供“发后不理”能力，让程序指定需要做什么，但怎么做这样的低层细节交给运行时完成。

这种观点从 `java.util.concurrent` 包含的并发工具包中可以窥探一二，本书不会详细介绍这个包，有兴趣的读者可以阅读 Brian Goetz 等人写的 *Java Concurrency in Practice* 一书（Addison-Wesley 出版）。

本章剩下的内容会介绍 Java 平台提供的低层并发机制，每个 Java 开发者都应该对此有所了解。

6.5.1 线程的生命周期

我们先来看看应用线程的生命周期。不同的操作系统看待线程的视角有所不同，因此在某些细节上可能有所不能（不过，站在一定高度上，大多数情况下基本类似）。Java 做了很多工作，力求把这些细节抽象化。Java 提供了一个名为 `Thread.State` 的枚举类型，囊括了操作系统看到的线程状态。`Thread.State` 中的值概述了一个线程的生命周期。

- `NEW`

  已经创建线程，但还没在线程对象上调用 `start()` 方法。所有线程一开始都处于这个状态。

- `RUNNABLE`

  线程正在运行，或者当操作系统调度线程时可以运行。

- BLOCKED

  线程中止运行，因为它在等待获得一个锁，以便进入声明为 synchronized 的方法或代码块。本节后面会详细介绍声明为 synchronized 的方法和代码块。

- WAITING

  线程中止运行，因为它调用了 Object.wait() 或 Thread.join() 方法。

- TIMED_WAITING

  线程中止运行，因为它调用了 Thread.sleep() 方法，或者调用了 Object.wait() 或 Thread.join() 方法，而且传入了超时时间。

- TERMINATED

  线程执行完毕。线程对象的 run() 方法正常退出，或者抛出了异常。

这些是常见的线程状态（至少对主流操作系统来说如此），线程的生命周期如图 6-4 所示。

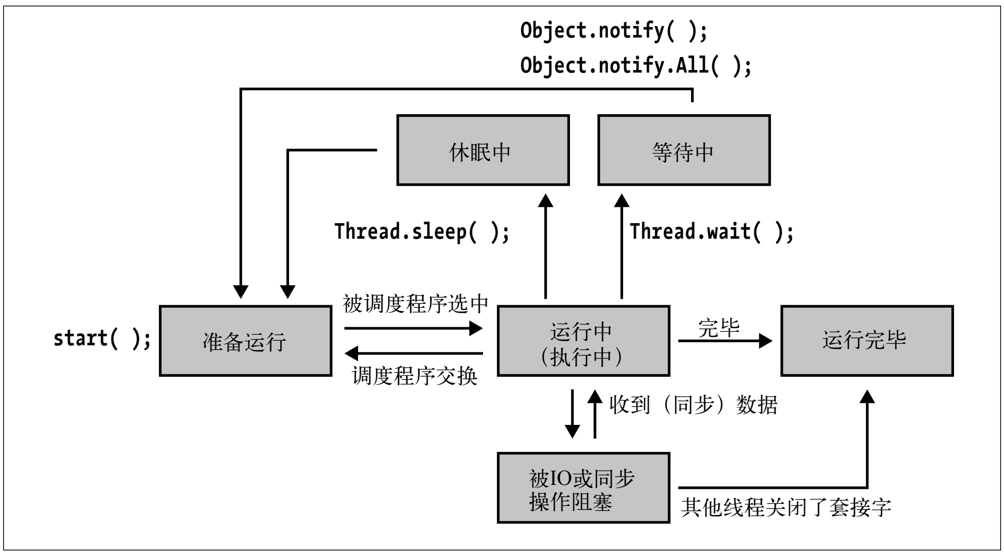


图 6-4：线程的生命周期

使用 Thread.sleep() 方法可以让线程休眠。这个方法有一个参数，指定线程休眠的时长，单位为毫秒，如下所示：

```
try {
    Thread.sleep(2000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

参数中指定的休眠时长是对操作系统的请求，而不是要求。例如，休眠的时间可能比请求的长。具体休眠多久，取决于负载和运行时环境相关的其他因素。

本章后面会介绍 `Thread` 类的其他方法，不过在此之前，我们要介绍一些重要的理论，学习线程如何访问内存，了解为什么多线程编程如此之难，会给开发者带来很多问题。

6.5.2　可见性和可变性

在 Java 中，其实一个进程中的每个 Java 应用线程都有自己的栈（和局部变量），不过这些线程共用同一个堆，因此可以轻易在线程之间共享对象，毕竟需要做的只是把引用从一个线程传到另一个线程，如图 6-5 所示。

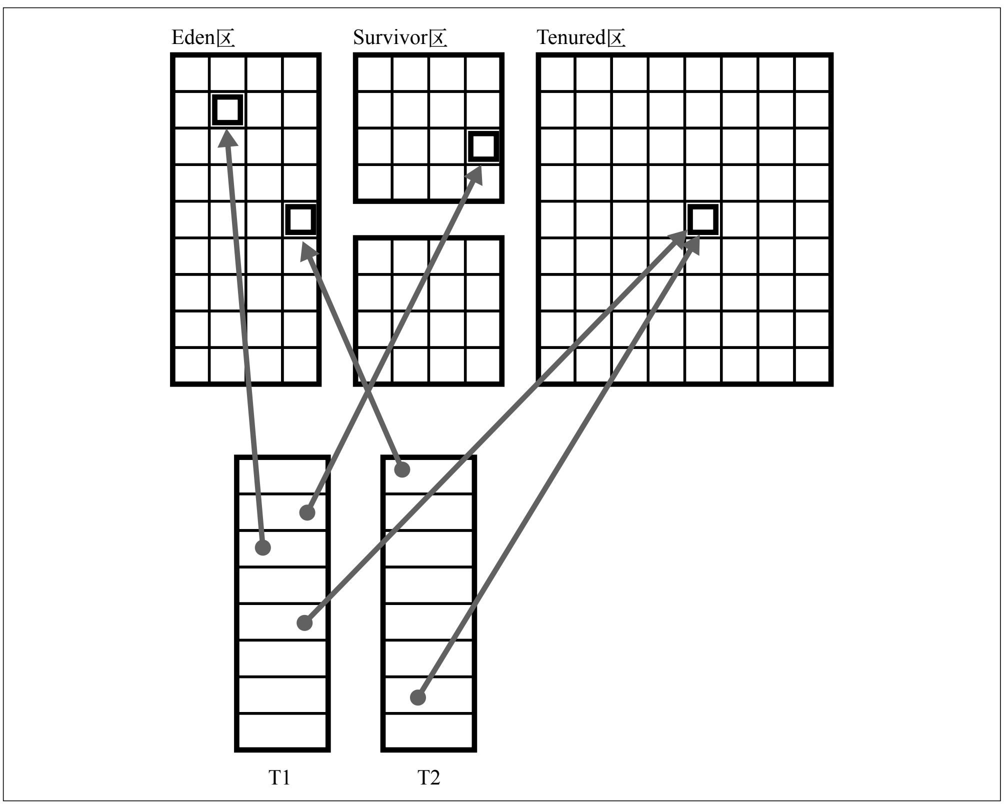


图 6-5：在线程之间共享内存

由此引出 Java 的一个一般设计原则——*对象默认可见*。如果我有一个对象的引用，就可以复制一个副本，然后将其交给另一个线程，不受任何限制。Java 中的引用其实就是类型指

针，指向内存中的一个位置，而且所有线程都共用同一个地址空间，所以默认可见符合自然规律。

除了默认可见之外，Java 还有一个特性对理解并发很重要——对象是*可变的*（mutable），对象的内容（实例字段的值）一般都可以修改。使用 `final` 关键字可以把变量或引用声明为常量，但这种字段不属于对象的内容。

在阅读本章剩下内容的过程中，我们会发现，这两个特性（跨线程可见性和对象可变性）结合在一起，大大增加了理解 Java 并发编程的难度。

并发编程的安全性

如果我们想编写正确的多线程代码，得让程序满足一个重要的条件，即：

> 在一个程序中，不管调用什么方法，也不管操作系统如何调度应用线程，一个对象看到的任何其他对象都不处于非法或不一致的状态，这样的程序才称得上是*安全的多线程程序*。

在第 5 章，我们把安全的面向对象程序定义为，通过调用对象的存取方法，把对象从一个合法状态变成另一个合法状态。这个定义对单线程代码来说没问题，但延伸到并发程序，会遇到一个特别的难题。

在大多数主流场合中，操作系统会根据负载和系统中运行的其他程序作出决策，在不同的时期把线程调度到不同的处理器内核中运行。如果负载高，说明还有其他进程需要运行。

如果需要，操作系统会把 Java 线程从 CPU 内核中强制移出，不管线程正在做什么，哪怕某个方法正执行一半，都会立即挂起。可是，第 5 章说过，在方法执行的过程中，可以临时先把对象变成非法状态，等方法退出后再变成合法状态。

因此，即便程序遵守了安全规则，如果一个长时间运行的方法还没退出线程就被踢出了，也可能会让对象处于不一致状态。也就是说，虽然为单线程正确建模了数据类型，还是要考虑如何避免并发的影响。添加这层额外的保护措施之后，才能称为*并发安全的*代码。

下一节介绍获取这层安全性的主要方式，本章末尾还会介绍在某些情况下有用的其他机制。

6.5.3　互斥和状态保护

只要修改或读取对象的过程中，对象的状态可能不一致，这段代码就要受到保护。为了保护这种代码，Java 平台只提供了一种机制：*互斥*。

假如一个方法包含一连串操作，那么在执行过程中中断，就可能会导致某个对象处于不一致或非法状态。如果这个非法状态对另一个对象可见，代码的行为可能就会错乱。

例如，在 ATM 或其他柜员机的系统中可能有如下代码：

```
public class Account {
    private double balance = 0.0; // 必须>= 0
    // 假设还有其他字段,例如name,以及其他方法
    // 例如deposit()、checkBalance()和dispenseNotes()

    public Account(double openingBal) {
        balance = openingBal;
    }

    public boolean withdraw(double amount) {
        if (balance >= amount) {
            try {
                Thread.sleep(2000); // 模拟风险检查
            } catch (InterruptedException e) {
                return false;
            }
            balance = balance - amount;
            dispenseNotes(amount);
            return true;
        }
        return false;
    }
}
```

`withdraw()` 方法中的一连串操作就可能会让对象处于不一致状态。具体来说是这样的，查看余额之后，在模拟风险检查阶段，第一个线程休眠时，可能会出现第二个线程继续执行代码，导致账户透支，违背 `balance >= 0` 这个约束条件。

在这个例子中，系统对对象的操作虽然在单线程中安全（因为在单线程中对象不可能变成非法状态，即 `balance < 0`），但并发时却不安全。

为了让这种代码在并发运行时也安全，Java 为开发者提供了 `synchronized` 关键字。这个关键字可以用在代码块或方法上，使用时，Java 平台会限制访问代码块或方法中的代码。

> 因为 `synchronized` 关键字把代码包围起来，所以很多开发者认为，Java 的并发和代码有关。有些资料甚至把 `synchronized` 修饰的块或方法中的代码称为临界区，还认为临界区是并发的关键所在。其实不然，稍后会看到，其实我们要防范的是数据的不一致性。

Java 平台会为它创建的每个对象记录一个特殊的标记，这个标记叫监视器（monitor）。`synchronized` 使用这些监视器（或叫锁）指明，随后的代码可以临时把对象渲染成不一致

的状态。synchronized 修饰的代码块或方法会发生一系列事件，详述如下：

(1) 线程需要修改对象时，会临时把对象变成不一致状态；

(2) 线程获取监视器，指明它需要临时互斥存储这个对象；

(3) 线程修改对象，修改完毕后对象处于一致的合法状态；

(4) 线程释放监视器。

如果在修改对象的过程中，其他线程尝试获取锁，Java 会阻塞这次尝试，直到拥有锁的线程释放锁为止。

注意，如果程序没有创建共享数据的多个线程，就无需使用 synchronized 语句。如果自始至终只有一个线程访问某个数据结构，就无需使用 synchronized 保护这个结构。

获取监视器不能避免访问对象，只能避免其他线程声称拥有这个锁——这一点至关重要。为了正确编写并发安全的代码，开发者要确保，修改或读取可能处于不一致状态的对象之前，得先获取对象的监视器。

换个角度来说，如果 synchronized 修饰的方法正在处理一个对象，并且把这个对象变成非法状态，那么读取这个对象的另一个方法（没使用 synchronized 修饰）仍能看到这个不一致的状态。

> 同步是保护状态的一种协助机制，因此非常脆弱。一个缺陷（需要使用 synchronized 修饰的方法却没有使用）就可能为系统的整体安全性带来灾难性的后果。

之所以使用 synchronized 这个词作为“需要临时互斥存储”的关键词，除了说明需要获取监视器之外，还表明进入代码块时，JVM 会从主内存中重新读取对象的当前状态。类似地，退出 synchronized 修饰的代码块或方法时，JVM 会刷新所有修改过的对象，把新状态存入主内存。

如果不同步，系统中不同的 CPU 内核看到的内存状态可能不一样，而这种差异可能会破坏运行中程序的状态。前面的 ATM 示例就可能出现这种情况。

6.5.4 volatile关键字

Java 还提供了另一个关键字，用来并发访问数据——volatile。这个关键字指明，应用代码使用字段或变量前，必须重新从主内存读取值。同样，修改使用 volatile 修饰的值后，在写入变量之后，必须存回主内存。

volatile 关键字的主要用途之一是在“关闭前一直运行”模式中使用。编写多线程程序时，如果外部用户或系统需要向处理中的线程发出信号，告诉线程在完成当前作业后优雅关闭线程，那么就要使用 volatile。这个过程有时叫作“优雅结束”模式。下面看个典型示例，假设处理中的线程里有下述代码，而这段代码在一个实现 Runnable 接口的类中定义：

```
private volatile boolean shutdown = false;

public void shutdown() {
    shutdown = true;
}

public void run() {
    while (!shutdown) {
        // ……处理其他任务
    }
}
```

只要没有其他线程调用 shutdown() 方法，处理中的线程就会继续处理任务（经常和非常有用的 BlockingQueue 一起使用，BlockingQueue 接口用于分配工作）。一旦有其他线程调用 shutdown() 方法，处理中的线程就会发现 shutdown 的值变成了 true。这个变化并不影响运行中的作业，不过一旦这个任务结束，处理中的线程就不会再接受其他任务，而会优雅关闭。

6.5.5 Thread类中有用的方法

创建新应用线程时，程序员可以使用 Thread 类中的许多方法，减少劳动量。这里没有列出全部方法，Thread 类还有一些其他方法，但本节主要介绍较常用的方法。

- getId()

这个方法返回线程的 ID 值，类型为 long。线程的 ID 在线程的整个生命周期中都不变。

- getPriority()和setPriority()

这两个方法控制线程的优先级。调度程序处理线程优先级的策略之一是，如果有优先级高的线程在等待，就不运行优先级低的线程。不过，大多数情况下都无法影响调度程序解释优先级的方式。线程的优先级使用 1~10 之间的整数表示。

- setName()和getName()

开发者使用这两个方法设定或取回单个线程的名称。为线程起名字是个好习惯，因为这样调试时更方便，尤其是使用 jvisualvm 等工具。13.2 节会介绍如何使用 jvisualvm。

- getState()

返回一个 Thread.State 对象，说明线程处于什么状态。表示状态的各个值在 6.5.1 节介绍过。

- isAlive()

用来测试线程是否还“活着”。

- start()

这个方法用来创建一个新应用线程，然后再调用 run() 方法调度这个线程，开始执行。正常情况下，执行到 run() 方法的末尾或者执行 run() 方法中的一个 return 语句后，线程就会结束运行。

- interrupt()

如果调用 sleep()、wait() 或 join() 方法时阻塞了某个线程，那么在表示这个线程的 Thread 对象上调用 interrupt() 方法，会让这个线程抛出 InterruptedException 异常（并把线程唤醒）。如果线程中涉及可中断的 I/O 操作，那么这个 I/O 操作会终止，而且线程会收到 ClosedByInterruptException 异常。即便线程没有从事任何可中断的操作，线程的中断状态也会被设为 true。

- join()

在调用 join() 方法的 Thread 对象“死亡”之前，当前线程一直处于等待状态。可以把这个方法理解为一个指令，在其他线程结束之前，当前线程不会继续向前运行。

- setDaemon()

*用户线程*是这样一种线程，只要它还“活着”，进程就无法退出——这是线程的默认行为。有时，程序员希望线程不阻止进程退出——这种线程叫*守护线程*。一个线程是守护线程还是用户线程，由 setDaemon() 方法控制。

- setUncaughtExceptionHandler()

线程因抛出异常而退出时，默认的行为是打印线程的名称、异常的类型、异常消息和堆栈跟踪。如果这么做还不够，可以在线程中安装一个自定义的处理程序，处理未捕获的异常。例如：

```
// 这个线程直接抛出一个异常
Thread handledThread =
  new Thread(() -> { throw new UnsupportedOperationException(); });

// 给线程起个名字,有利于调试
handledThread.setName("My Broken Thread");

// 处理这个异常的处理程序
handledThread.setUncaughtExceptionHandler((t, e) -> {
    System.err.printf("Exception in thread %d '%s':" +
        "%s at line %d of %s%n",
        t.getId(),    // 线程的ID
        t.getName(),  // 线程的名称
        e.toString(), // 异常名称和消息
        e.getStackTrace()[0].getLineNumber(),
```

```
            e.getStackTrace()[0].getFileName()); });
    handledThread.start();
```

这个方法在某些情况下很有用，例如，如果一个线程在监管一组其他工作线程，那么可以使用这种模式重启“死亡”的线程。

Thread类弃用的方法

Thread 类除了有一些有用的方法之外，还有一些危险的方法，开发者不应该使用。这些方法是 Java 线程 API 原来提供的，但很快就发现不适合开发者使用。可惜的是，因为 Java 要向后兼容，所以不能把这些方法从 API 中移除。开发者要知道有这些方法，而且在任何情况下都不能使用。

- stop()

如若不违背并发安全的要求，几乎不可能正确使用 Thread.stop()，因为 stop() 方法会立即“杀死”线程，不会给线程任何机会把对象恢复成合法状态。这和并发安全等原则完全相悖，因此绝对不能使用 stop() 方法。

- suspend()、resume()和countStackFrames()

调用 suspend() 方法挂起线程时，不会释放这个线程拥有的任何一个监视器，因此，如果其他线程试图访问这些监视器，这些监视器会变成死锁。其实，这种机制会导致死锁之间的条件竞争，而且 resume() 会导致这几个方法不能使用。

- destroy()

这个方法一直没有实现，如果实现了，会遇到与 suspend() 方法一样的条件竞争。

开发者始终应该避免使用这些弃用的方法。为了达到上述方法的预期作用，Java 开发了一些安全的替代模式。前面提到的“关闭前一直运行”模式就是这些模式的一例。

6.6 使用线程

若想有效使用多线程代码，要对监视器和锁有些基本的认识。你需要知道的要点如下。

- 同步是为了保护对象的状态和内存，而不是代码。
- 同步是线程间的协助机制。一个缺陷就可能破坏这种协助模型，导致严重的后果。
- 获取监视器只能避免其他线程再次获取这个监视器，而不能保护对象。
- 即便对象的监视器锁定了，不同步的方法也能看到（和修改）不一致的状态。
- 锁定 Object[] 不会锁定其中的单个对象。
- 基本类型的值不可变，因此不能（也无需）锁定。
- 接口中声明的方法不能使用 synchronized 修饰。
- 内部类只是语法糖，因此内部类的锁对外层类无效（反过来亦然）。

- Java 的锁可重入（reentrant）。这意味着，如果一个线程拥有一个监视器，这个线程遇到具有同一个监视器的同步代码块时，可以进入这个代码块。[2]

我们还说过，线程可以休眠一段时间。但有时不需要指定具体休眠多久，而是等到满足某个条件时才唤醒。在 Java 中，这种操作通过 wait() 和 notify() 方法完成，这两个方法都在 Object 类中定义。

就像每个 Java 对象都关联一个锁一样，每个对象还会维护一个等待线程列表。在一个线程中，如果某个对象调用了 wait() 方法，那么这个线程会临时释放它拥有的所有锁，而且这个线程会被添加到这个对象的等待线程列表中，然后停止运行。其他线程在这个对象上调用 notifyAll() 方法时，这个对象会唤醒等待线程，让这些线程继续运行。

例如，下面是一个简化版队列，在多线程环境中可以安全使用：

```
/*
 * 一个线程调用push()方法,把一个对象存入队列。
 * 另一个线程调用pop()方法,从队列中取出一个对象。
 * 如果队列中没有数据,pop()方法使用wait()/notify(),一直等待,直到有数据。
 */
public class WaitingQueue<E> {
    LinkedList<E> q = new LinkedList<E>(); // 仓库
    public synchronized void push(E o) {
        q.add(o);         // 把对象添加到链表的末端
        this.notifyAll(); // 告诉等待的线程,数据准备好了
    }
    public synchronized E pop() {
        while(q.size() == 0) {
            try { this.wait(); }
            catch (InterruptedException ignore) {}
        }
        return q.remove();
    }
}
```

这个类在队列为空时（此时 pop() 操作会失败）在 WaitingQueue 实例上调用 wait() 方法。等待的线程会临时释放监视器，允许其他线程声称拥有这个监视器，然后这个线程可能会调用 push() 方法，把新对象添加到队列中。原来的线程被唤醒时，会从它之前开始休眠的地方继续运行，而且会重新获取监视器。

> wait() 和 notify() 方法必须在 synchronized 修饰的方法或代码块中使用，因为只有临时把锁放弃，这两个方法才能正常工作。

注 2：除了 Java，其他语言实现的锁并不都有这种特性。

一般来说，大多数开发者都不需要自己编写类似这个示例的类，使用 Java 平台提供的库和组件即可。

6.7 小结

本章介绍了 Java 实现内存管理和并发编程的方式，以及这两个话题之间的内在联系。处理器的内核数量越来越多，因此我们要使用并发编程技术合理利用这些内核。未来，并发决定着应用的性能。

Java 的线程模型基于三个基本概念。

- *状态是共享的，可变的，而且默认可见*

  意思是，在同一个进程中，对象可在不同的线程间轻易共享，而且只要线程中有对象的引用，就可以修改对象。

- *抢先式线程调度*

  几乎任何时候，操作系统的线程调度程序都能把线程调入和调出内核。

- *对象的状态只能由锁保护*

  锁很难正确使用，而且状态十分脆弱，即便是读取操作也可能会得到不可思议的结果。

Java 实现并发的这三个方面放在一起，解释了为什么多线程编程会让开发者如此头痛。

第二部分

使用Java平台

第二部分介绍 Java 原生的一些核心库，以及中高级 Java 程序员常用的一些编程技术。

- 第 7 章　编程和文档约定
- 第 8 章　使用 Java 集合
- 第 9 章　处理常见的数据格式
- 第 10 章　处理文件和 I/O
- 第 11 章　类加载、反射和方法句柄
- 第 12 章　Nashorn
- 第 13 章　平台工具和配置

第 7 章

编程和文档约定

本章说明一些重要且有用的 Java 编程和文档约定，包含以下内容：

- 一般的命名和大小写约定
- 可移植性的技巧和约定
- 文档注释 javadoc 的句法和约定

7.1 命名和大小写约定

下述广泛采用的命名约定适用于 Java 中的包、引用类型、方法、字段和常量。这些约定几乎全球通用，而且会影响你定义的类的公开 API，因此要认真遵守。

- 包

  公开可见的包通常要尽量使用唯一的包名。十分常见的做法是，把网站的域名倒过来，放在包名前（例如 com.oreilly.javanutshell）。所有包名都应该使用小写字母。

  如果包只在应用内部使用，而且打包成 JAR 文件分发，那么这种包公开不可见，无需遵守上述约定。此时，经常使用应用的名称作包名或包名的前缀。

- 引用类型

  类型的名称应该以大写字母开头，而且要混用大小写（例如 String）。如果类名包含多个单词，每个单词的第一个字母都要大写（例如 StringBuffer）。如果类型名称或类型名称中有一部分是简称，那么简称可以全用大写字母（例如 URL 和 HTMLParser）。

  类和枚举类型是为了表示对象，因此类名要使用名词（例如 Thread、Teapot 和

FormatConverter）。

如果接口是用来为实现这个接口的类提供额外信息的，那么一般应该使用形容词命名这个接口（例如 Runnable、Cloneable 和 Serializable）。注解类型一般也使用这种命名方式。

如果接口的作用更像是抽象超类，那么应该使用名词命名（例如 Document，FileNameMap 和 Collection）。

- *方法*

  方法名始终以小写字母开头。如果方法名包含多个单词，除第一个单词外，其他单词的第一个字母都要大写（例如 insert()、insertObject() 和 insertObjectAt()）。这种命名方式一般称为"驼峰式"。

  方法名一般都经过精心挑选，让第一个单词为动词。为了清楚表明方法的作用，方法名的长度不限，但应该尽量选择简短的名称。应该避免使用太通用的方法名，例如 performAction()、go() 或糟透的 doIt()。

- *字段和常量*

  非常量字段的名称使用的大小写约定和方法名一样。如果是声明为 static final 的常量，其名称应该使用全大写形式。如果常量的名称包含多个单词，单词之间应该使用下划线分隔（例如 MAX_VALUE）。为字段选择的名称，应该最能说明字段或其值的作用。枚举类型定义的常量往往也全部使用大写字母。

- *参数*

  方法的参数使用的大小写约定和非常量字段一样。方法的参数名会出现在方法的文档中，因此应该选择一个能尽量清楚表明参数作用的名称。尽量使用一个单词命名参数，并在所有用到这个参数的地方使用相同的名称。例如，如果 WidgetProcessor 类定义的方法中，有多个方法的第一个参数都是一个 Widget 对象，那么在每个方法中可以都把这个参数命名为 widget 甚或 w。

- *局部变量*

  局部变量的名称是实现细节，在类外部不可见。尽管如此，选择一个好名称仍会让代码更易于阅读、理解和维护。变量的命名方式往往与方法和字段的命名约定一样。

除了名称的种类有专门的约定之外，名称中可以使用的字符也有约定。Java 虽然允许在标识符中使用 $ 字符，但按照约定，$ 专门用于源代码处理程序生成的合成名称。例如，Java 编译器使用 $ 字符实现内部类。在你起的任何名称中都不应该使用 $ 字符。

Java 允许名称使用 Unicode 字符集中的任何字母数字字符。这对不说英语的程序员而言虽然便利，但始终没有流行开来，极少见到有人这么做。

7.2 实用的命名方式

我们为结构起的名字十分重要。向同事表述我们的抽象构思时，命名是一个关键。把软件构思从一个人的头脑中转移到另一个人的头脑中很难，多数情况下，甚至比转移到实现构思的机器中还难。

因此，我们必须竭尽所能，把这个过程变得简单易行。而名称是关键所在。审查代码时（所有代码都应该审查），审查人员应该特别留意代码中使用的名称。

- 类型的名称能否表明类型的作用？
- 各个方法所做的事情是否完全和方法名表达的意思一致？理想情况下，应该不多也不少。
- 名称的表述是否到位？要不要换成更具体的名称？
- 名称是否适用于所描述的领域？
- 同一领域中使用的名称是否一致？
- 名称中是否混杂着隐喻？
- 名称是否重用了软件工程常用的术语？

混杂隐喻在软件中很常见，尤其是应用发布几版之后。一开始，系统的组件可能会使用完全合理的名称，例如 `Receptionist`（处理进入的连接）、`Scribe`（持久存储订单）和 `Auditor`（检查和调整订单），但很快，在下一版中就会出现一个名为 `Watchdog` 的类，用于重启进程。这样命名并不糟糕，但破坏了以前建立起来的命名模式——以职务头衔取名。

你要意识到，随着时间的推移，软件经常会变动。这一点非常重要。版本 1 中使用的名称完全贴切，但在版本 4 中可能会变得含糊不清。注意，随着系统关注点和意图的变化，重构代码时也要重构名称。现代化 IDE 可以全局搜索并替换符号，因此不必固守不再适用的过时隐喻。

最后提醒一下，过度严格地解读这些规则，可能会导致开发者使用非常奇怪的命名结构。如果一成不变地使用这些约定，可能会导致一些荒唐的结果，有些资料对此做了生动的描述。

换句话说，这里所述的约定，没有一条是强制要求。如果遵守，绝多大多数情况下都能让代码变得更易于阅读和维护。不过，你可能还记得乔治·奥威尔的一句名言：“宁愿打破这些规则，也不说任何不着调的话。”因此，如果能让代码更易于阅读，别害怕打破这些准则。

最重要的是，你要对你编写的代码能存活多久有个理性认识。银行的风险计算系统可能要使用十年或更久，而初创项目的原型可能只会存在几周时间。因此，你要根据代码的生存时间相应地编写文档，代码存在的时间越长，文档就要写得越好。

7.3 Java文档注释

Java 代码中的多数普通注释是用来说明代码实现细节的。不过，Java 语言规范还定义了一种特殊的注释，叫*文档注释*（doc comment），这种注释用于编写代码 API 的文档。

文档注释是普通的多行注释，以 /** 开头（不是通常使用的 /*），以 */ 结尾。文档注释放在类型或成员定义的前面，其中的内容是那个类型或成员的文档。文档中可以包含简单的 HTML 格式化标签，还可以包含其他特殊的关键字，提供额外的信息。编译器会忽略文档注释，但 javadoc 程序能把文档注释提取出来，自动转换成 HTML 格式的在线文档（javadoc 的更多信息参见第 13 章）。下面这个示例定义一个类，而且包含适当的文档注释：

```
/**
 * 这个不可变的类表示<i>复数</i>
 *
 * @author David Flanagan
 * @version 1.0
 */
public class Complex {
    /**
     * 保存复数的实部
     * @see #y
     */
    protected double x;

    /**
     * 保存复数的虚部
     * @see #x
     */
    protected double y;

    /**
     * 创建一个新Complex对象,表示复数x+yi
     * @param x 复数的实部
     * @param y 复数的虚部
     */
    public Complex(double x, double y) {
        this.x = x;
        this.y = y;
    }

    /**
     * 两个Complex对象相加,然后创建第三个对象,表示二者之和
     * @param c1 一个Complex对象
     * @param c2 另一个Complex对象
     * @return  一个新Complex对象,表示<code>c1</code>和<code>c2</code>之和
     * @exception java.lang.NullPointerException
     *            如果有一个参数是<code>null</code>
     */
    public static Complex add(Complex c1, Complex c2) {
        return new Complex(c1.x + c2.x, c1.y + c2.y);
```

```
    }
}
```

7.3.1 文档注释的结构

文档注释主体的开头是一句话，概述类型或成员的作用。这句话可能会在文档的概述中显示，因此应该自成一体。第一句话后面可以跟着其他句子或段落，数量不限，这些内容用来详细说明类、接口、方法或字段。

在这些描述性段落之后，还可以有其他段落，数量也不限，而且每段都以一个特殊的文档注释标签开头，例如 `@author`、`@param` 或 `@returns`。这些包含标签的段落提供类、接口、方法或字段的特殊信息，`javadoc` 程序会以一种标准的方式显示这些信息。全部文档注释标签在下一节列出。

文档注释的描述性内容可以包含简单的 HTML 标记标签，例如：`<i>` 用于强调，`<code>` 用于显示类、方法和字段的名称，`<pre>` 用于显示多行代码示例。除此之外，也可以包含 `<p>` 标签，把说明分成多个段落；还可以使用 `<ul>` 和 `<li>` 等相关标签，显示无序列表等结构。不过，要记住，你编写的内容会嵌入复杂的大型 HTML 文档，因此，文档注释不能包含 HTML 主结构标签，例如 `<h2>` 和 `<hr>`，以防影响那个大型 HTML 文档的结构。

在文档注释中，应该避免使用 `<a>` 标签加入超链接或交叉引用。如果有这方面的需求，应该使用特殊的文档注释标签 `{@link}`。这个标签和其他文档注释标签不同，可以在文档注释的任何位置使用。下一节会介绍，`{@link}` 标签的作用是插入超链接，指向其他类、接口、方法或字段，但无需知道 `javadoc` 程序使用的 HTML 结构约定和文件名。

如果想在文档注释中插入图片，要把图片文件放在源码目录里的 doc-files 子目录中，而且要使用类名和一个整数后缀命名这个图片。例如，`Circle` 类文档注释中的第二张图片，可以使用下述 HTML 标签插入：

```
<img src="doc-files/Circle-2.gif">
```

文档注释中的各行都嵌在一个 Java 注释里，因此，处理之前，每一行注释前面的空格和星号都会删掉。所以，无需担心星号会出现在生成的文档中，也无需担心注释的缩进会影响 `<pre>` 标签中代码示例的缩进。

7.3.2 文档注释标签

`javadoc` 程序能识别一些特殊的标签，每个标签都以 `@` 字符开头。这些文档注释标签以一种标准的方式在注释中插入特殊的信息，`javadoc` 会根据所用的标签选择合适的格式输出信息。例如，`@param` 标签用于指定方法中一个参数的名称和意义。`javadoc` 会把这些信息提取出来，视情况而定，将其显示在 HTML 的 `<dl>` 列表或 `<table>` 表格中。

javadoc 能识别的文档注释标签如下所示，文档注释一般应该按照下述顺序使用这些标签。

- `@author name`

  添加一个“Author:”条目，内容是指定的名字。每个类和接口定义都应该使用这个标签，但单个方法和字段一定不能使用。如果一个类有多位作者，在相邻的几行中使用多个 `@author` 标签。例如：

  ```
  @author Ben Evans
  @author David Flanagan
  ```

  多位作者按照时间顺序列出，先列出最初的作者。如果不知道作者是谁，可以使用“unascribed”[1]。如果不指定命令行参数 `-author`，`javadoc` 不会输出作者信息。

- `@version text`

  插入一个“Version:”条目，内容是指定的文本。例如：

  ```
  @version 1.32, 08/26/04
  ```

  每个类和接口的文档注释中都应该包含这个标签，但单个方法和字段不能使用。这个标签经常和支持自动排序版本号的版本控制系统一起使用，例如 git、Perforce 或 SVN。如果不指定命令行参数 `-versio`，`javadoc` 不会输出版本信息。

- `@param parameter-name description`

  把指定的参数及其说明添加到当前方法的“Parameters:”区域。在方法和构造方法的文档注释中，每个参数都要使用一个 `@param` 标签列出，而且应该按照参数传入方法的顺序排列。这个标签只能出现在方法或构造方法的文档注释中。

  鼓励使用短语和句子片段，保持说明简洁。不过，如果需要详细说明参数，说明文字可以分成多行，需要多少字就写多少字。为了在源码中易于阅读，可以使用空格对齐所有说明。例如：

  ```
  @param o       要插入的对象
  @param index   插入对象的位置
  ```

- `@return description`

  插入一个“Returns:”区域，内容是指定的说明。每个方法的文档注释中都应该使用这个标签，除非方法返回 `void`，或者是构造方法。说明需要多长就可以写多长，但为了保持简短，建议使用句子片段。例如：

  ```
  @return <code>true</code>：成功插入
          <code>false</code>：列表中已经包含要插入的对象
  ```

注 1：这个单词可以理解为中文里的“佚名”。——译者注

- `@exception full-classname description`

  添加一个“Throws:”条目，内容是指定的异常名称和说明。方法和构造方法的文档注释应该为 `throws` 子句中的每个已检异常编写一个 `@exception` 标签。例如：

  ```
  @exception java.io.FileNotFoundException
             如果找不到指定的文件
  ```

  如果方法的用户基于某种原因想捕获当前方法抛出的未检异常（即 `RuntimeException` 的子类），`@exception` 标签也可以为这些未检异常编写文档。如果方法能抛出多个异常，要在相邻的几行使用多个 `@exception` 标签，而且按照异常名称的字母表顺序排列。根据需要，说明可长可短，只要能说清异常的意思就行。这个标签只能出现在方法和构造方法的文档注释中。`@throws` 标签是 `@exception` 标签的别名。

- `@throws full-classname description`

  这个标签是 `@exception` 标签的别名。

- `@see reference`

  添加一个“See Also:”条目，内容是指定的引用。这个标签可以出现在任何文档注释中。引用的句法在 7.3.4 节说明。

- `@deprecated explanation`

  这个标签指明随后的类型或成员弃用了，应该避免使用。`javadoc` 会在文档中添加一个明显的“Deprecated”条目，内容为指定的 `explanation` 文本。这个文本应该说明这个类或成员从何时开始弃用，如果可能的话，还要推荐替代的类或成员，并且添加指向替代的类或成员的链接。例如：

  ```
  @deprecated 从3.0版开始,这个方法被{@link #setColor}取代了。
  ```

  一般情况下，`javac` 会忽略所有注释，但 `@deprecated` 标签是个例外。如果文档注释中有这个标签，编译器会在生成的类文件中注明弃用信息，提醒其他类，这个功能已经弃用。

- `@since version`

  指明类型或成员何时添加到 API 中。这个标签后面应该跟着版本号或其他形式的版本信息。例如：

  ```
  @since JNUT 3.0
  ```

  每个类型的文档注释都应该包含一个 `@since` 标签；类型初始版本之后添加的任何成员，都要在其文档注释中加上 `@since` 标签。

- `@serial description`

  严格来说，类序列化的方式是公开 API 的一部分。如果你编写的类可以序列化，就应该在文档注释中使用 `@serial` 标签和下面列出的相关标签说明序列化的格式。在实现 `Serializable` 接口的类中，组成序列化状态的每个字段，都应该在其文档注释中使用 `@serial` 标签。

  对于使用默认序列化机制的类来说，除了声明为 `transient` 的字段，其他所有字段，包括声明为 `private` 的字段，都要在文档注释中使用 `@serial` 标签。`description` 应该简要说明字段及其在序列化对象中的作用。

  在类和包的文档注释中也可以使用 `@serial` 标签，指明是否为当前类或包生成“Serialized Form”页面。句法如下：

  ```
  @serial include
  @serial exclude
  ```

- `@serialField name type description`

  实现 `Serializable` 接口的类可以声明一个名为 `serialPersistentFields` 的字段，定义序列化格式。`serialPersistentFields` 字段的值是一个数组，由 `ObjectStreamField` 对象组成。对这样的类来说，在 `serialPersistentFields` 字段的文档注释里，数组中的每个元素都要使用一个 `@serialField` 标签列出，每个标签都要指明元素在类序列化状态中的名称、类型和作用。

- `@serialData description`

  实现 `Serializable` 接口的类可以定义一个 `writeObject()` 方法，用于写入数据，代替默认序列化机制提供的写入方式。实现 `Externalizable` 接口的类可以定义一个 `writeExternal()` 方法，把对象的完整状态写入序列化流。`writeObject()` 和 `writeExternal()` 方法的文档注释中应该使用 `@serialData` 标签，`description` 应该说明这个方法使用的序列化格式。

7.3.3 行内文档注释标签

除了上述标签外，`javadoc` 还支持几个行内标签。在文档注释中，只要能使用 HTML 文本的地方都可以使用行内标签。因为这些标签直接出现在 HTML 文本流中，所以要使用花括号把标签中的内容和周围的 HTML 文本隔开。`javadoc` 支持的行内标签包括如下几个。

- `{@link reference }`

  `{@link}` 标签和 `@see` 标签的作用类似，但 `@see` 标签是在专门的“See Also:”区域放一个指向引用的链接，而 `{@link}` 标签在行内插入链接。在文档注释中，只要能使用 HTML 文本的地方都可以使用 `{@link}` 标签。因此，`{@link}` 标签可以出现在类、接口、

方法或字段的第一句话中，也能出现在 @param、@returns、@exception 和 @deprecated 标签的说明中。{@link} 标签中的 reference 使用专门的句法，7.3.4 节会介绍。例如：

```
@param regexp 搜索时使用的正则表达式。这个字符串参数使用的句法必须符合{@link java.
util.regex.Pattern}制定的规则。
```

- {@linkplain reference }

  {@linkplain} 标签和 {@link} 标签的作用类似，不过，在 {@linkplain} 标签生成的链接中，链接文字使用普通的字体，而 {@link} 标签使用代码字体。如果 reference 包含要链接的 feature 和指明链接替代文本的 label，就要使用 {@linkplain} 标签。7.3.4 节会讨论 reference 参数中的 feature 和 label 两部分。

- {@inheritDoc}

  如果一个方法覆盖了超类的方法，或者实现了接口中的方法，那么这个方法的文档注释可以省略一些内容，让 javadoc 自动从被覆盖或被实现的方法中继承。{@inheritDoc} 标签可以继承单个标签的文本，还能在继承的基础上再添加一些说明。继承单个标签的方式如下：

```
@param index @{inheritDoc}
@return @{inheritDoc}
```

- {@docRoot}

  这个行内标签没有参数，javadoc 生成文档时会把它替换成文档的根目录。这个标签在引用外部文件的超链接中很有用，例如引用一张图片或者一份版权声明：

```
<img src="{@docroot}/images/logo.gif">
这份资料受<a href="{@docRoot}/legal.html">版权保护</a>。
```

- {@literal text }

  这个行内标签按照字面形式显示 text，text 中的所有 HTML 都会转义，而且所有 javadoc 标签都会被忽略。虽然不保留空白格式，但仍适合在 <pre> 标签中使用。

- {@code text }

  这个标签和 {@literal} 标签的作用类似，但会使用代码字体显示 text 的字面量。等价于：

```
&lt;code&gt;{@literal <replaceable>text</replaceable>}&lt;/code&gt;
```

- {@value}

  没有参数的 {@value} 标签在 static final 字段的文档注释中使用，会被替换成当前字段的常量值。

- `{@value reference }`

  这种 `{@value}` 标签的变体有一个 `reference` 参数，指向一个 `static final` 字段，会被替换成指定字段的常量值。

7.3.4 文档注释中的交叉引用

`@see` 标签以及行内标签 `{@link}`、`{@linkplain}` 和 `{@value}` 都可以创建指向文档中其他内容的交叉引用，而且往往指向其他类型或成员的文档注释。

`reference` 参数有三种不同的格式。如果 `reference` 以引号开头，表示书名或其他出版物的名称，参数的值是什么就显示什么。如果 `reference` 以 < 符号开头，表示使用 `<a>` 标签标记的任意 HTML 超链接，这个超链接会原封不动地插入生成的文档。`@see` 标签使用这种形式插入指向其他在线文档的链接，例如程序员指南或用户手册。

如果 `reference` 既不是放在引号中的字符串，也不是超链接，那么应该具有下述格式：

```
feature [label]
```

此时，`javadoc` 会把 `label` 当成超链接的文本，指向 `feature` 指定的内容。如果没指定 `label`（一般都不指定），`javadoc` 会使用 `feature` 作为超链接的文本。

`feature` 可以指向包、类型或类型的成员，使用下述格式中的一种。

- `pkgname`

  指向指定的包。例如：

  ```
  @see java.lang.reflect
  ```

- `pkgname.typename`

  指定完整的包名，指向对应的类、接口、枚举类型或注解类型。例如：

  ```
  @see java.util.List
  ```

- `typename`

  不指定包名，指向对应的类型。例如：

  ```
  @see List
  ```

  `javadoc` 会搜索当前包和 `typename` 类导入的所有类，解析这个引用。

- `typename#methodname`

  指向指定类型中指定名称对应的方法或构造方法。例如：

  ```
  @see java.io.InputStream#reset
  ```

```
@see InputStream#close
```

如果类型不包含包名，会按照 typename 使用的方式解析。如果方法重载了，或类中定义有同名字段，这种句法会引起歧义。

- typename#methodname(paramtypes)

  指向某个方法或构造方法，而且明确指定参数的类型。交叉引用重载的方法时可以使用这种格式。例如：

```
@see InputStream#read(byte[], int, int)
```

- #methodname

  指向一个没有重载的方法或构造方法，这个方法在当前类或接口中，或者在当前类或接口的某个外层类、超类或超接口中。这种简短格式用于指向同一个类中的其他方法。例如：

```
@see #setBackgroundColor
```

- #methodname(paramtypes)

  指向当前类、接口或者某个超类、外层类中的方法或构造方法。这种格式可以指向重载的方法，因为它明确列出了方法参数的类型。例如：

```
@see #setPosition(int, int)
```

- typename#fieldname

  指向指定类中的指定字段。例如：

```
@see java.io.BufferedInputStream#buf
```

  如果类型不包含包名，会按照 typename 使用的方式解析。

- #fieldname

  指向一个字段，这个字段在当前类型中，或者在当前类型的某个外层类、超类或超接口中。例如：

```
@see #x
```

7.3.5　包的文档注释

类、接口、方法、构造方法和字段的文档注释放在这些结构的定义体之前。javadoc 也能读取并显示包的概述文档。包在一个目录中定义，而不是在单个源码文件中定义，因此，javadoc 会在包所在的目录（存放包中各个类的源码）中需找一个名为 package.html 的文件，这个文件中的内容就是包的文档。

package.html 文件可以包含简单的 HTML 格式文档，也可以使用 `@see`、`@link`、`@deprecated` 和 `@since` 标签。因为 package.html 不是 Java 源码文件，所以其中的文档应该是 HTML，而不能是 Java 注释（即不能包含在 `/**` 和 `*/` 之间）。最后，在 package.html 文件中，所有 `@see` 和 `@link` 标签都必须使用完全限定的类名。

除了可以为每个包定义 package.html 文件之外，还可以为一组包提供概括性文档，方法是在这组包所在的源码树中创建一个 overview.html 文件。`javadoc` 解析这个源码树时，会提取 overview.html 文件中的内容，作为最高层概览显示出来。

7.4　可移植程序的约定

Java 最早使用的宣传语之一是：“一次编写，到处运行。”这个宣传语强调了，使用 Java 可以轻松写出可移植的程序，但 Java 程序仍然有可能无法自动在所有 Java 平台中成功运行。下述技巧有助于避免移植性问题。

- *本地方法*

  可移植的 Java 代码可以使用 Java 核心 API 中的任何方法，包括本地方法。但是，在可移植的代码中不能定义本地方法。就其本质而言，本地方法必须移植到每一种新平台中，因此直接违背了 Java“一次编写，到处运行”的承诺。

- `Runtime.exec()`*方法*

  可移植的代码很少允许调用 `Runtime.exec()` 方法派生进程，或者在本地系统中执行外部命令，因为无法保证执行的操作系统本地命令在所有平台中都存在或表现一致。在可移植的代码中只有一种情况能使用 `Runtime.exec()` 方法——允许用户指定要执行的命令，可以在运行时输入，也可以在配置文件或首选项对话框中指定。

- `System.getenv()`*方法*

  使用 `System.getenv()` 方法的代码一定不可移植。

- *没有文档的类*

  可移植的 Java 代码只能使用 Java 平台中有文档的类和接口。多数 Java 实现都包含了一些没有文档的公开类，这些类虽是实现的一部分，但不是 Java 平台规范的一部分。没什么能阻止程序使用并依赖这些没有文档的类，但这么做可能会导致程序不可移植，因为无法保证所有 Java 实现和所有平台中都有这些类。

  在这些类中要特别注意 `sun.misc.Unsafe` 类，这个类提供了一些“不安全”的方法，可以让开发者避开 Java 平台的一些重要限制。在任何情况下，开发者都不应该直接使用 `Unsafe` 类。

- `java.awt.peer`包

  `java.awt.peer` 包中的接口是 Java 平台的一部分，但其文档只说明了如何在 AWT 系统中使用。直接使用这些接口的应用不可移植。

- 某个实现特有的特性

  可移植的代码绝对不能依赖某个实现特有的特性。例如，微软提供了一个 Java 运行时系统，这个系统包含一些 Java 平台规范中没有定义的方法。使用这些扩展功能的程序显然不能移植到其他平台。

- 某个实现特有的缺陷

  就像不能依赖某个实现特有的特性一样，可移植的代码也绝对不能依赖某个实现特有的缺陷。如果类或方法的表现和规范中所述的有所不同，可移植的程序就不能依赖这种行为，因为在不同的平台可能有不同的表现，而且最终可能会被修复。

- 某个实现特有的行为

  有时，不同的平台和不同的实现会有不同的行为，根据 Java 规范，这种差异是合法的。可移植的代码绝对不能依赖某种特定的行为。例如，Java 规范没有规定具有相同优先级的程序能否共享 CPU，也没有规定长时间运行的线程能不能排挤具有相同优先级的其他线程。如果应用假定某种行为，可能无法在全部平台中正常运行。

- 标准扩展

  可移植的代码可以依赖 Java 平台的标准扩展，不过，如果这么做，要清楚地指出用了哪些扩展，而且在没有安装这些扩展的系统中运行时要输出适当的错误消息，利落地退出。

- 完整的程序

  所有可移植的 Java 程序都必须是完整的，而且要自成一体：除了核心平台和标准扩展类之外，必须提供用到的所有类。

- 定义系统类

  可移植的 Java 代码决不能在任何系统包或标准扩展包中定义类。这么做会破坏包的保护界线，而且会暴露包可见的实现细节。

- 硬编码文件名

  可移植的程序不能使用硬编码的文件名或目录名，因为不同的平台使用十分不同的文件系统组织方式，而且使用不同的目录分隔符。如果要使用文件或目录，让用户指定文件名，至少也要让用户指定文件所在的基目录。这个操作可在运行时完成，在配置文件或程序的命令行参数中指定文件名。需要把文件名或目录名连接到目录名上时，要使用 `File()` 构造方法或 `File.separator` 常量。

- 换行符

  不同的系统使用不同的字符或字符序列做换行符。在程序中不要把换行符硬编码成 `\n`、`\r` 或 `\r\n`，而要使用 `PrintStream` 或 `PrintWriter` 类中的 `println()`，这个方法换行时会自动使用适用于当前平台的换行符，或者使用系统属性 `line.separator` 的值也行。在 `java.util.Formatter` 及相关类的 `printf()` 和 `format()` 方法中，还可以使用“%n”格式化字符串。

第 8 章

使用Java集合

本章介绍 Java 支持的一种基本数据结构——Java 集合。集合是很多（也可能是多数）编程方式的抽象，是程序员基本工具包的重要组成部分。因此，本章是全书最重要的章节之一，几乎所有 Java 程序员都要了解这些知识。

本章介绍基本的接口和类型层次结构，说明怎么使用这些接口，还会讨论总体设计要略。本章还涵盖处理集合的“经典”方式和全新方式（使用 Java 8 引入的流 API 和 lambda 表达式）。

8.1 介绍集合API

Java 集合是一系列泛型接口，描述最常见的数据类型格式。Java 为每一种典型的数据结构都提供了多种实现方式，而且这些类型都通过接口实现，因此开发团队可以自行开发专用的实现方式，在自己的项目中使用。

Java 集合定义了两种基本的数据结构，一种是 `Collection`，表示一组对象的集合；另一种是 `Map`，表示对象间的一系列映射或关联关系。Java 集合的基本架构如图 8-1 所示。

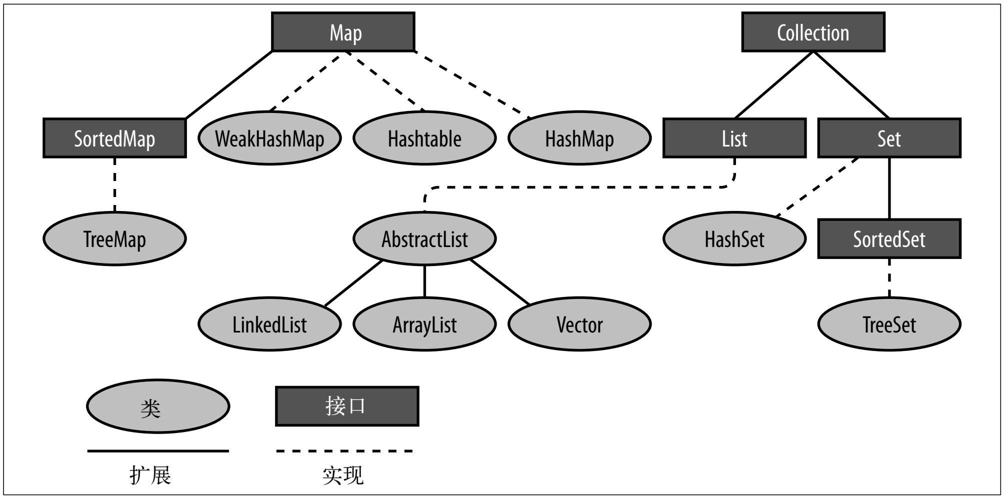


图 8-1：集合类及其继承关系

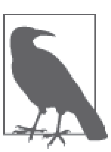
在这种架构中，`Set` 是一种 `Collection`，不过其中没有重复的对象；`List` 也是一种 `Collection`，其中的元素按顺序排列（不过可能有重复）。

`SortedSet` 和 `SortedMap` 是特殊的集和映射，其中的元素按顺序排列。

`Collection`、`Set`、`List`、`Map`、`SortedSet` 和 `SortedMap` 都是接口，不过 `java.util` 包定义了多个具体实现，例如基于数组和链表的列表，基于哈希表或二叉树的映射和集。除此之外，还有两个重要的接口，`Iterator` 和 `Iterable`，用于遍历集合中的对象，稍后会介绍。

8.1.1 Collection接口

`Collection<E>` 是参数化接口，表示由泛型 `E` 对象组成的集合。这个接口定义了很多方法，用来把对象添加到集合中，把对象从集合中移除，测试对象是否在集合中，以及遍历集合中的所有元素。还有一些方法可以把集合中的元素转换成数组，以及返回集合的大小。



> `Collection` 集合可以允许其中的元素有重复，也可以禁止有重复；可以强制使用特定的顺序，也可以不强制有顺序。

Java 集合框架之所以提供 `Collection` 接口，是因为常见的数据结构都会用到这些特性。JDK 提供的 `Set`、`List` 和 `Queue` 都是 `Collection` 的子接口。下述代码展示了可以在 `Collection` 对象上执行的操作：

```
// 创建几个集合,供后面的代码使用
Collection<String> c = new HashSet<>(); // 一个空集

// 稍后会介绍这些实用方法
// 注意,使用时要留意一些细节
Collection<String> d = Arrays.asList("one", "two");
Collection<String> e = Collections.singleton("three");

// 向集合中添加一些元素
// 如果集合的内容变化了,这些方法返回true
// 这种表现对不允许重复的Set类型很有用
c.add("zero");           // 添加单个元素
c.addAll(d);             // 添加d中的所有元素

// 复制集合:多数实现都有副本构造方法
Collection<String> copy = new ArrayList<String>(c);

// 把元素从集合中移除。
// 除了clear()方法之外,如果集合的内容变化了,都返回true
c.remove("zero");        // 移除单个元素
c.removeAll(e);          // 移除一组元素
c.retainAll(d);          // 移除不在e中的所有元素
c.clear();               // 移除所有元素

// 获取集合的大小
boolean b = c.isEmpty(); // c是空的,所以返回true
int s = c.size();        // 现在c的大小是0

// 使用前面创建的副本复原集合
c.addAll(copy);

// 测试元素是否在集合中。测试基于equals()方法,而不是==运算符
b = c.contains("zero");  // true
b = c.containsAll(d);    // true

// 多数Collection实现都有toString()方法
System.out.println(c);

// 使用集合中的元素创建一个数组。
// 如果迭代器能保证特定的顺序,数组就有相同的顺序
// 得到的数组是个副本,而不是内部数据结构的引用
Object[] elements = c.toArray();

// 如果想把集合中的元素存入String[]类型的数组,必须在参数中指定这个类型
String[] strings = c.toArray(new String[c.size()]);

// 或者传入一个类型为String[]的空数组,指定所需的类型
// toArray()方法会为这个数组分配空间
strings = c.toArray(new String[0]);
```

记住，上述各个方法都能用于 Set、List 或 Queue。这几个子接口可能会对集合中的元素做些限制或有顺序上的约束，但都提供了相同的基本方法。

修改集合的方法，例如 add()、remove()、clear() 和 retainAll()，是可选的 API。不过，这个规则在很久以前就定下了，那时认为如果不提供这些方法，明智的做法是抛出 UnsupportedOperationException 异常。因此，某些实现（尤其是只读方法）可能会抛出未检异常。

Collection 和 Map 及其子接口都没扩展 Cloneable 或 Serializable 接口。不过，在 Java 集合框架中，实现集合和映射的所有类都实现了这两个接口。

有些集合对其可以包含的元素做了限制。例如，有的集合禁止使用 null 作为元素。EnumSet 要求其中的元素只能是特定的枚举类型。

如果尝试把禁止使用的元素添加到集合中，会抛出未检异常，例如 NullPointerException 或 ClassCastException。检查集合中是否包含禁止使用的元素，可能也会抛出这种异常，或者仅仅返回 false。

8.1.2　Set接口

集（set）是无重复对象组成的集合：不能有两个引用指向同一个对象，或两个指向 null 的引用，如果对象 a 和 b 的引用满足条件 a.equals(b)，那么这两个对象也不能同时出现在集中。多数通用的 Set 实现都不会对元素排序，但并不禁止使用有序集（SortedSet 和 LinkedHashSet 就有顺序）。而且集与列表等有序集合不同，一般认为，集的 contains 方法，不论以常数时间还是以对数时间评判[1]，运行效率都高。

除了 Collection 接口定义的方法之外，Set 没有再定义其他方法，但对这些方法做了额外的限制。Set 接口要求 add() 和 addAll() 方法必须遵守无重复原则：如果集中已经有某个元素，就不能再次添加。前面说过，Collection 接口定义的 add() 和 addAll() 方法，在改变集合时返回 true，否则返回 false。对 Set 对象而言，也会返回 true 或 false，因为不能重复意味着添加元素不一定会修改集。

表 8-1 列出了实现 Set 接口的类，而且总结了各个类的内部表示方式、排序特性、对成员的限制，以及 add()、remove()、contains 等基本操作和迭代的性能。这些类的详细信息，请参见各自的文档。注意，CopyOnWriteArraySet 类在 java.util.concurrent 包中，其他类则在 java.util 包中。还要注意，java.util.BitSet 类没有实现 Set 接口，这个类过时了，用于紧凑而高效地表示布尔值组成的列表，但不是 Java 集合框架的一部分。

注 1：常数时间和对数时间是定量描述算法运行时间的两种方式。——译者注

表8-1：实现Set接口的类

类	内部表示	首次出现的版本	元素顺序	成员限制	基本操作	迭代性能	备　注
`HashSet`	哈希表	1.2	无	无	$O(1)$	O(capacity)	最佳通用实现
`LinkedHashSet`	哈希链表	1.2	插入的顺序	无	$O(1)$	$O(n)$	保留插入的顺序
`EnumSet`	位域	5.0	枚举声明	枚举类型的值	$O(1)$	$O(n)$	只能保存不是 `null` 的枚举值
`TreeSet`	红黑树	1.2	升序排列	可比较	$O(\log(n))$	$O(n)$	元素所属的类型要实现 `Comparable` 或 `Comparator` 接口
`CopyOnWriteArraySet`	数组	5.0	插入的顺序	无	$O(n)$	$O(n)$	不使用同步方法也能保证线程安全

`TreeSet` 类使用红黑树数据结构维护集，这个集中的元素按照 `Comparable` 对象的自然顺序升序迭代，或者按照 `Comparator` 对象指定的顺序迭代。其实，`TreeSet` 实现的是 `Set` 的子接口，`SortedSet` 接口。

`SortedSet` 接口提供了多个有趣的方法，这些方法都考虑到了元素是有顺序的，如下述代码所示：

```
public static void testSortedSet(String[] args) {
    // 创建一个SortedSet对象
    SortedSet<String> s = new TreeSet<>(Arrays.asList(args));

    // 迭代集:元素已经自动排序
    for (String word : s) {
        System.out.println(word);
    }

    // 特定的元素
    String first = s.first(); // 第一个元素
    String last = s.last();   // 最后一个元素

    // 除第一个元素之外的其他所有元素
    SortedSet<String> tail = s.tailSet(first + '\0');
    System.out.println(tail);

    // 除最后一个元素之外的其他所有元素
    SortedSet<String> head = s.headSet(last);
    System.out.println(head);

    SortedSet<String> middle = s.subSet(first+'\0', last);
    System.out.println(middle);
}
```

必须加上 \0 字符，因为 tailSet() 等方法要使用某个元素后面的元素，对字符串来说，要在后面加上 NULL 字符（对应于 ASCII 中的 0）。

8.1.3 List接口

List 是一组有序的对象集合。列表中的每个元素都有特定的位置，而且 List 接口定义了一些方法，用于查询或设定特定位置（或叫*索引*）的元素。从这个角度来看，List 对象和数组类似，不过列表的大小能按需变化，以适应其中元素的数量。和集不同，列表允许出现重复的元素。

除了基于索引的 get() 和 set() 方法之外，List 接口还定义了一些方法，用于把元素添加到特定的索引，把元素从特定的索引移除，或者返回指定值在列表中首次出现或最后出现的索引。从 Collection 接口继承的 add() 和 remove() 方法，前者把元素添加到列表末尾，后者把指定值从列表中首次出现的位置移除。继承的 addAll() 方法把指定集合中的所有元素添加到列表的末尾，或者插入指定的索引。retainAll() 和 removeAll() 方法的表现与其他 Collection 对象一样，如果需要，会保留或删除多个相同的值。

List 接口没有定义操作索引范围的方法，但是定义了一个 subList() 方法。这个方法返回一个 List 对象，表示原列表指定范围内的元素。子列表会回馈父列表，只要修改了子列表，父列表立即就能察觉到变化。下述代码演示了 subList() 方法和其他操作 List 对象的基本方法：

```
// 创建两个列表,供后面的代码使用
List<String> l = new ArrayList<String>(Arrays.asList(args));
List<String> words = Arrays.asList("hello", "world");

// 通过索引查询和设定元素
String first = l.get(0);        // 列表的第一个元素
String last = l.get(l.size -1); // 列表的最后一个元素
l.set(0, last);                 // 把最后一个元素变成第一个元素

// 添加和插入元素
// add()方法既可以把元素添加到列表末尾,也可以把元素插入指定索引
l.add(first);         // 把第一个词添加到列表末尾
l.add(0, first);      // 再把第一个词添加到列表的开头
l.addAll(words);      // 把一个集合添加到列表末尾
l.addAll(1, words);   // 在第一个词之后插入一个集合

// 子列表:回馈原列表
List<String> sub = l.subList(1,3); // 第二个和第三个元素
sub.set(0, "hi");                  // 修改l的第二个元素
// 子列表可以把操作限制在原列表索引的子范围内
String s = Collections.min(l.subList(0,4));
```

```
Collections.sort(l.subList(0,4));
// 子列表的独立副本不影响父列表
List<String> subcopy = new ArrayList<String>(l.subList(1,3));

// 搜索列表
int p = l.indexOf(last); // 最后一个词在哪个位置？
p = l.lastIndexOf(last); // 反向搜索

// 打印last在l中出现的所有索引。注意,使用了子列表
int n = l.size();
p = 0;
do {
    // 创建一个列表,只包含尚未搜索的元素
    List<String> list = l.subList(p, n);
    int q = list.indexOf(last);
    if (q == -1) break;
    System.out.printf("Found '%s' at index %d%n", last, p+q);
    p += q+1;
} while(p < n);

// 从列表中删除元素
l.remove(last);         // 把指定元素从首次出现的位置上删除
l.remove(0);            // 删除指定索引对应的元素
l.subList(0,2).clear(); // 使用subList()方法,删除一个范围内的元素
l.retainAll(words);     // 删除所有不在words中的元素
l.removeAll(words);     // 删除所有在words中的元素
l.clear();              // 删除所有元素
```

1. 遍历循环和迭代

操作集合有种重要的方式：依次处理每个元素，这种方式叫迭代。这种处理数据结构的方式很古老，但时至今日依旧很有用（尤其是用来处理少量数据的集合），而且易于理解。迭代最适合使用 for 循环，而且使用 List 对象演示最简单，如下所示：

```
ListCollection<String> c = new ArrayList<String>();
// ……把一些字符串添加到c中
for(String word : c) {
    System.out.println(word);
}
```

这段代码的作用很明显——一次读取 c 中的一个元素，然后把这个元素当成变量传入循环主体。说得更正式一些，这段代码的作用是迭代数组或集合（或者其他实现 java.lang.Iterable 接口的对象）中的元素。每次迭代都会把数组或 Iterable 对象中的一个元素赋值给你声明的循环变量，然后执行循环主体，一般都会处理表示元素的循环变量。迭代的过程中不需要使用循环计数器或 Iterator 对象，遍历循环会自动迭代，你无需担心初始化或终止循环的方式是否正确。

这种 for 循环一般称作遍历循环。我们来看一下它的运作方式。下述代码重写了前面的 for 循环（作用等效），显示了迭代过程中真正会使用的方法：

```
// 使用for循环迭代
for(Iterator<String> i = c.iterator(); i.hasNext();) {
    System.out.println(i.next());
}
```

Iterator 对象 i 从集合上获取，用于逐个读取集合中的元素。在 while 循环中也可以使用 Iterator 对象：

```
// 使用while循环迭代集合中的元素
// 有些集合种类(例如列表)能保障迭代的顺序,有些则不能
Iterator<String> iterator = c.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
```

关于遍历循环的句法，还有一些注意事项。

- 前面说过，expression 必须是数组或实现 java.lang.Iterable 接口的对象。编译时必须知道 expression 的类型，这样才能生成合适的循环代码。
- 数组或 Iterable 对象中元素的类型必须与 declaration 中声明的变量类型兼容，这样才能赋值。如果使用的 Iterable 对象没有使用元素的类型参数化，那么变量必须声明为 Object 类型。
- declaration 一般只包含变量的类型和名称，不过也可以包含 final 修饰符和任何适当的注解（参见第 4 章）。final 的作用是避免循环变量使用循环赋予它的数组或集合元素之外的值，以此强调不能通过循环变量修改数组或集合。
- 遍历循环的循环变量的声明必须是循环的一部分，变量的类型和名称都要指明。不能像 for 循环那样，使用循环之外声明的变量。

为了深入理解遍历循环处理集合的方式，我们要了解两个接口——java.util.Iterator 和 java.lang.Iterable：

```
public interface Iterator<E> {
     boolean hasNext();
     E next();
     void remove();
}
```

Iterator 接口定义了一种迭代集合或其他数据结构中元素的方式。迭代的过程是这样的：只要集合中还有更多的元素（hasNext() 方法返回 true），就调用 next() 方法获取集合中的下一个元素。有序集合（例如列表）的迭代器一般能保证按照顺序返回元素。无序集合（例如 Set）只能保证不断调用 next() 方法返回集中的所有元素，没有遗漏也没有重复，不过没有特定的顺序。

> Iterator 接口中的 next() 方法有两个作用：把集合的游标向前移动，然后返回集合的前一个头值。如果使用不可变的风格编程，这两个操作可能导致问题，因为 next() 其实会修改集合。

引入 Iterable 接口是为了支持遍历循环。类实现这个接口是为了表明它提供了 Iterator 对象，可以迭代：

```
public interface Iterable<E> {
    java.util.Iterator<E> iterator();
}
```

如果某个对象是 Iterable<E> 类型，表明它有一个能返回 Iterator<E> 对象的 iterator() 方法，而 Iterator<E> 对象有一个 next() 方法，能返回一个 E 类型的对象。

> 注意，如果使用遍历循环迭代 Iterable<E> 对象，循环变量必须是 E 类型，或者是 E 类型的超类或实现的接口。

例如，迭代 List<String> 对象中的元素时，循环变量必须声明为 String 类型或超类 Object 类型，或者 String 实现的某个接口：CharSequence、Comparable 或 Serializable。

2. 随机访问列表中的元素

我们一般期望实现 List 接口的类能高效迭代，而且所用时间和列表的大小成正比。然而，不是所有列表都能高效地随机访问任意索引上的元素。按顺序访问的列表，例如 LinkedList 类，提供了高效的插入和删除操作，但降低了随机访问性能。提供高效随机访问的类都实现了标记接口 RandomAccess，因此，如果需要确定是否能高效处理列表，可以使用 instanceof 运算符测试是否实现了这个接口：

```
// 随便创建一个列表,供后面的代码处理
List<?> l = ...;

// 测试能否高效随机访问
// 如果不能,先使用副本构造方法创建一个支持高效随机访问的副本,然后再处理
if (!(l instanceof RandomAccess)) l = new ArrayList<?>(l);
```

在 List 对象上调用 iterator() 方法会得到一个 Iterator 对象，这个对象按照元素在列表中的顺序迭代各元素。List 实现了 Iterable 接口，因此列表可以像其他集合一样使用遍历循环迭代。

如果只想迭代列表中的部分元素，可以使用 subList() 方法创建子列表：

```
List<String> words = ...; // 创建一个列表,供下面的代码迭代

// 迭代除第一个元素之外的所有元素
for(String word : words.subList(1, words.size ))
    System.out.println(word);
```

表 8-2 总结了 Java 平台中五种通用的 List 实现。Vector 和 Stack 类已经过时，别再用了。CopyOnWriteArrayList 类在 java.util.concurrent 包中，只适合在多线程环境中使用。

表8-2：实现List接口的类

类	表示方式	首次出现的版本	随机访问	备　注
ArrayList	数组	1.2	能	最佳全能实现
LinkedList	双向链表	1.2	否	高效插入和删除
CopyOnWriteArrayList	数组	5.0	能	线程安全；遍历快，修改慢
Vector	数组	1.0	能	过时的类；同步的方法。不要使用
Stack	数组	1.0	能	扩展 Vector 类；添加了 push()、pop() 和 peek() 方法。过时了，用 Deque 替代

8.1.4　Map接口

映射（map）是一系列键值对，一个键对应一个值。Map 接口定义了用于定义和查询映射的 API。Map 接口属于 Java 集合框架，但没有扩展 Collection 接口，因此 Map 只是一种集合，而不是 Collection 类型。Map 是参数化类型，有两个类型变量。类型变量 K 表示映射中键的类型，类型变量 V 表示键对应的值的类型。例如，如果有个映射，其键是 String 类型，对应的值是 Integer 类型，那么这个映射可以表示为 Map<String,Integer>。

Map 接口定义了几个最有用的方法：put() 方法定义映射中的一个键值对，get() 方法查询指定键对应的值，remove() 方法把指定的键及对应的值从映射中删除。一般来说，实现 Map 接口的类都要能高效执行这三个基本方法：一般应该运行在常数时间中，而且绝不能比在对数时间中运行的性能差。

Map 的重要特性之一是，可以视作集合。虽然 Map 对象不是 Collection 类型，但映射的键可以看成 Set 对象，映射的值可以看成 Collection 对象，而映射的键值对可以看成由 Map.Entry 对象组成的 Set 对象。（Map.Entry 是 Map 接口中定义的嵌套接口，表示一个键值对。）

下述示例代码展示了如何使用 get()、put() 和 remove() 等方法操作 Map 对象，还演示了把 Map 对象视作集合后的一些常见用法：

```
// 新建一个空映射
Map<String,Integer> m = new HashMap();

// 不可变的映射,只包含一个键值对
```

```
Map<String,Integer> singleton = Collections.singletonMap("test", -1);

// 注意,极少使用下述句法显式指定通用方法emptyMap()的参数类型
// 得到的映射不可变
Map<String,Integer> empty = Collections.<String,Integer>emptyMap();

// 使用put()方法填充映射,把数组中的元素映射到元素的索引上
String[] words = { "this", "is", "a", "test" };
for(int i = 0; i < words.length; i++) {
    m.put(words[i], i); // 注意,int会自动装包成Integer
}

// 一个键只能映射一个值
// 不过,多个键可以映射同一个值
for(int i = 0; i < words.length; i++) {
    m.put(words[i].toUpperCase(), i);
}

// putAll()方法从其他映射中复制键值对
m.putAll(singleton);

// 使用get()方法查询映射
for(int i = 0; i < words.length; i++) {
    if (m.get(words[i]) != i) throw new AssertionError();
}

// 测试映射中是否有指定的键和值
m.containsKey(words[0]);       // true
m.containsValue(words.length); // false

// 映射的键、值和键值对都可以看成集合
Set<String> keys = m.keySet();
Collection<Integer> values = m.values();
Set<Map.Entry<String,Integer>> entries = m.entrySet();

// 映射和上述几个集合都有有用的toString()方法
System.out.printf("Map: %s%nKeys: %s%nValues: %s%nEntries: %s%n",
                  m, keys, values, entries);

// 可以迭代这些集合
// 多数映射都没定义迭代的顺序(不过SortedMap定义了)
for(String key : m.keySet()) System.out.println(key);
for(Integer value: m.values()) System.out.println(value);

// Map.Entry<K,V>类型表示映射中的一个键值对
for(Map.Entry<String,Integer> pair : m.entrySet()) {
    // 打印键值对
    System.out.printf("'%s' ==> %d%n", pair.getKey(), pair.getValue());
    // 然后把每个Entry对象的值增加1
    pair.setValue(pair.getValue() + 1);
}

// 删除键值对
m.put("testing", null);   // 映射到null上可以"擦除"一个键值对
m.get("testing");         // 返回null
```

```
m.containsKey("testing"); // 返回true:键值对仍然存在
m.remove("testing");      // 删除键值对
m.get("testing");         // 还是返回null
m.containsKey("testing"); // 这次返回false

// 也可以把映射视作集合,然后再删除
// 不过,向集合中添加键值对时不能这么做
m.keySet().remove(words[0]); // 等同于m.remove(words[0]);

// 删除一个值为2的键值对——这种方式一般效率不高,用途有限
m.values().remove(2);
// 删除所有值为4的键值对
m.values().removeAll(Collections.singleton(4));
// 只保留值为2和3的键值对
m.values().retainAll(Arrays.asList(2, 3));

// 还可以通过迭代器删除
Iterator<Map.Entry<String,Integer>> iter = m.entrySet().iterator();
while(iter.hasNext()) {
    Map.Entry<String,Integer> e = iter.next();
    if (e.getValue() == 2) iter.remove();
}

// 找出两个映射中都有的值
// 一般来说,addAll()和retainAll()配合keySet()和values()使用,可以获取交集和并集
Set<Integer> v = new HashSet<Integer>(m.values());
v.retainAll(singleton.values());

// 其他方法
m.clear();                // 删除所有键值对
m.size();                 // 返回键值对的数量:目前为0
m.isEmpty();              // 返回true
m.equals(empty);          // true:实现Map接口的类覆盖了equals()方法
```

Map 接口有一些通用和专用的实现，表 8-3 对此做了总结。和之前一样，完整的细节参见 JDK 文档和 javadoc。在表 8-3 中，除了 ConcurrentHashMap 和 ConcurrentSkipListMap 两个类在 java.util.concurrent 包中，其他类都在 java.util 包中。

表8-3：实现Map接口的类

类	表示方式	首次出现的版本	null键	null值	备　注
HashMap	哈希表	1.2	是	是	通用实现
ConcurrentHashMap	哈希表	5.0	否	否	通用的线程安全实现；参见 ConcurrentMap 接口
ConcurrentSkipListMap	哈希表	6.0	否	否	专用的线程安全实现；参见 Concurrent-NavigableMap 接口
EnumMap	数组	5.0	否	是	键是枚举类型
LinkedHashMap	哈希表加列表	1.4	是	是	保留插入或访问顺序

（续）

类	表示方式	首次出现的版本	null键	null值	备　　注
TreeMap	红黑树	1.2	否	是	按照键排序。操作耗时为 $O(\log(n))$。参见 SortedMap 接口
IdentityHashMap	哈希表	1.4	是	是	比较时使用 ==，而不使用 equals()
WeakHashMap	哈希表	1.2	是	是	不会阻止垃圾回收键
Hashtable	哈希表	1.0	否	否	过时的类；同步的方法。不要使用
Properties	哈希表	1.0	否	否	使用 String 类的方法扩展 Hashtable 接口

java.util.concurrent 包中的 ConcurrentHashMap 和 ConcurrentSkipListMap 两个类实现了同一个包中的 ConcurrentMap 接口。ConcurrentMap 接口扩展 Map 接口，而且定义了一些对多线程编程很重要的原子操作方法。例如，putIfAbsent() 方法，它的作用和 put() 方法类似，不过，仅当指定的键没有映射到其他值上时，才会把键值对添加到映射中。

TreeMap 类实现 SortedMap 接口。这个接口扩展 Map 接口，添加了一些方法，利用这种映射的有序特性。SortedMap 接口和 SortedSet 接口非常相似。firstKey() 和 lastKey() 方法分别返回 keySet() 所得集的第一个和最后一个键。而 headMap()、tailMap() 和 subMap() 方法都返回一个新映射，由原映射特定范围内的键值对组成。

8.1.5　Queue接口和BlockingQueue接口

队列（queue）是一组有序的元素，提取元素时按顺序从队头读取。队列一般按照插入元素的顺序实现，因此分成两类：先进先出（first-in, first-out，FIFO）队列和后进先出（last-in, first-out，LIFO）队列。

> LIFO 队列也叫栈（stack），Java 提供了 Stack 类，但强烈不建议使用——应该使用实现 Deque 接口的类。

队列也可以使用其他顺序：优先队列（priority queue）根据外部 Comparator 对象或 Comparable 类型元素的自然顺序排序元素。与 Set 不同的是，Queue 的实现往往允许出现重复的元素。而与 List 不同的是，Queue 接口没有定义处理任意索引位元素的方法，只有队列的头一个元素能访问。Queue 的所有实现都要具有一个固定的容量：队列已满时，不能再添加元素。类似地，队列为空时，不能再删除元素。很多基于队列的算法都会用到满和空这两个状态，所以 Queue 接口定义的方法通过返回值表明这两个状态，而不会抛出异常。具体而言，peek() 和 poll() 方法返回 null 表示队列为空。因此，多数 Queue 接口的实现不允许用 null 作元素。

阻塞式队列（blocking queue）是一种定义了阻塞式 put() 和 take() 方法的队列。put() 方法的作用是把元素添加到队列中，如果需要，这个方法会一直等待，直到队列中有存储元素的空间为止。而 take() 方法的作用是从队头移除元素，如果需要，这个方法会一直等待，直到队列中有元素可供移除为止。阻塞式队列是很多多线程算法的重要组成部分，因此 BlockingQueue 接口（扩展 Queue 接口）在 java.util.concurrent 包中定义。

队列不像集、列表和映射那么常用，只在特定的多线程编程风格中会用到。这里，我们不举实例，而是试着厘清一些令人困惑的队列插入和移除操作。

1. 把元素添加到队列中

- add()方法

  这个方法在 Collection 接口中定义，只是使用常规的方式添加元素。对有界的队列来说，如果队列已满，这个方法可能会抛出异常。

- offer()方法

  这个方法在 Queue 接口中定义，但是由于有界的队列已满而无法添加元素时，这个方法返回 false，而不会抛出异常。

  BlockingQueue 接口定义了一个超时版 offer() 方法，如果队列已满，会在指定的时间内等待空间。这个版本和基本版一样，成功插入元素时返回 true，否则返回 false。

- put()方法

  这个方法在 BlockingQueue 接口中定义，会阻塞操作：如果因为队列已满而无法插入元素，put() 方法会一直等待，直到其他线程从队列中移除元素，有空间插入新元素为止。

2. 把元素从队列中移除

- remove()方法

  Collection 接口中定义了 remove() 方法，把指定的元素从队列中移除。除此之外，Queue 接口还定义了一个没有参数的 remove() 方法，移除并返回队头的元素。如果队列为空，这个方法会抛出 NoSuchElementException 异常。

- poll()方法

  这个方法在 Queue 接口中定义，作用和 remove() 方法类似，移除并返回队头的元素，不过，如果队列为空，这个方法会返回 null，而不抛出异常。

  BlockingQueue 接口定义了一个超时版 poll() 方法，在指定的时间内等待元素添加到空队列中。

- take()方法

  这个方法在 BlockingQueue 接口中定义，用于删除并返回队头的元素。如果队列为空，这个方法会等待，直到其他线程把元素添加到队列中为止。

- `drainTo()`*方法*

  这个方法在 `BlockingQueue` 接口中定义，作用是把队列中的所有元素都移除，然后把这些元素添加到指定的 `Collection` 对象中。这个方法不会阻塞操作，等待有元素添加到队列中。这个方法有个变体，接受一个参数，指定最多移除多少个元素。

3. 查询

就队列而言，“查询”的意思是访问队头的元素，但不将其从队列中移除。

- `element()`*方法*

  这个方法在 `Queue` 接口中定义，其作用是返回队头的元素，但不将其从队列中移除。如果队列为空，这个方法抛出 `NoSuchElementException` 异常。

- `peek()`*方法*

  这个方法在 `Queue` 接口中定义，作用和 `element()` 方法类似，但队列为空时，返回 `null`。

> 使用队列时，最好选定一种处理失败的方式。例如，如果想在操作成功之前一直阻塞，应该选择 `put()` 和 `take()` 方法；如果想检查方法的返回值，判断操作是否成功，应该选择 `offer()` 和 `poll()` 方法。

`LinkedList` 类也实现了 `Queue` 接口，提供的是无界 FIFO 顺序，插入和移除操作需要常数时间。`LinkedList` 对象可以使用 `null` 作元素，不过，当列表用作队列时不建议使用 `null`。

`java.util` 包中还有另外两个 `Queue` 接口的实现。一个是 `PriorityQueue` 类，这种队列根据 `Comparator` 对象排序元素，或者根据 `Comparable` 类型元素的 `compareTo()` 方法排序元素。`PriorityQueue` 对象的队头始终是根据指定排序方式得到的最小值。另外一个是 `ArrayDeque` 类，实现的是双端队列，一般在需要用到栈的情况下使用。

`java.util.concurrent` 包中也包含一些 `BlockingQueue` 接口的实现，目的是在多线程编程环境中使用。有些实现很高级，甚至无需使用同步方法。

8.1.6 实用方法

`java.util.Collections` 类定义了一些静态实用方法，用于处理集合。其中有一类方法很重要，是集合的*包装*方法：这些方法包装指定的集合，返回特殊的集合。包装集合的目的是把集合本身没有提供的功能绑定到集合上。包装集合能提供的功能有：线程安全性、写保护和运行时类型检查。包装集合都以原来的集合为后盾，因此，在包装集合上调用的方法其实会分派给原集合的等效方法完成。这意味着，通过包装集合修改集合后，改动也会体现在原集合身上；反之亦然。

第一种包装方法为包装的集合提供线程安全性。java.util 包中的集合实现，除了过时的 Vector 和 Hashtable 类之外，都没有 synchronized 方法，不能禁止多个线程并发访问。如果需要使用线程安全的集合，而且不介意同步带来的额外开销，可以像下面这样创建集合：

```
List<String> list =
    Collections.synchronizedList(new ArrayList<String>());
Set<Integer> set =
    Collections.synchronizedSet(new HashSet<Integer>());
Map<String,Integer> map =
    Collections.synchronizedMap(new HashMap<String,Integer>());
```

第二种包装方法创建的集合对象不能修改底层集合，得到的集合是只读的，只要试图修改集合的内容，就会抛出 UnsupportedOperationException 异常。如果要把集合传入方法，但不允许修改集合，也不允许使用任何方式改变集合的内容，就可以使用这种包装集合：

```
List<Integer> primes = new ArrayList<Integer>();
List<Integer> readonly = Collections.unmodifiableList(primes);
// 可以修改primes列表
primes.addAll(Arrays.asList(2, 3, 5, 7, 11, 13, 17, 19));
// 但不能修改只读的包装集合
readonly.add(23); // 抛出UnsupportedOperationException异常
```

java.util.Collections 类还定义了用来操作集合的方法。其中最值得关注的是排序和搜索集合元素的方法：

```
Collections.sort(list);
// 必须先排序列表中的元素
int pos = Collections.binarySearch(list, "key");
```

Collections 类中还有些方法值得关注：

```
// 把list2中的元素复制到list1中,覆盖list1
Collections.copy(list1, list2);
// 使用对象o填充list
Collections.fill(list, o);
// 找出集合c中最大的元素
Collections.max(c);
// 找出集合c中最小的元素
Collections.min(c);

Collections.reverse(list); // 反转列表
Collections.shuffle(list); // 打乱列表
```

你最好全面熟悉 Collections 和 Arrays 类中的实用方法，这样遇到常见任务时就不用自己动手实现了。

特殊的集合

除了包装方法之外，java.util.Collections 类还定义了其他实用方法，一些用于创建只包含一个元素的不可变集合实例，一些用于创建空集合。singleton()、singletonList() 和

singletonMap() 方法分别返回不可变的 Set、List 和 Map 对象，而且只包含一个指定的对象或键值对。如果要把单个对象当成集合传入方法，可以使用这些方法。

Collections 类还定义了一些返回空集合的方法。如果你编写的方法要返回一个集合，遇到没有返回值的情况时，一般最好返回空集合，而不要返回 null 等特殊的值：

```
Set<Integer> si = Collections.emptySet();
List<String> ss = Collections.emptyList();
Map<String,Integer> m = Collections.emptyMap();
```

最后还有个 nCopies() 方法。这个方法返回一个不可变的 List 对象，包含指定数量个指定对象的副本：

```
List<Integer> tenzeros = Collections.nCopies(10, 0);
```

8.1.7　数组和辅助方法

由对象组成的数组和集合的作用类似，而且二者之间可以相互转换：

```
String[] a ={ "this", "is", "a", "test" }; // 一个数组
// 把数组当成大小不可变的列表
List<String> l = Arrays.asList(a);
// 创建一个大小可变的副本
List<String> m = new ArrayList<String>(l);

// asList()是个变长参数方法,所以也可以这么做：
Set<Character> abc = new HashSet<Character>(Arrays.asList('a', 'b', 'c'));

// Collection接口定义了toArray()方法。不传入参数时,这个方法创建
// Object[]类型的数组,把集合中的元素复制到数组中,然后返回这个数组
// 把set中的元素存入数组
Object[] members = set.toArray();
// 把list中的元素存入数组
Object[] items = list.toArray();
// 把map的键存入数组
Object[] keys = map.keySet().toArray();
// 把map的值存入数组
Object[] values = map.values().toArray();

// 如果不想返回Object[]类型的值,可以把一个所需类型的数组传入toArray()方法
// 如果传入的数组不够大,会再创建一个相同类型的数组
// 如果传入的数组太大,复制集合元素后剩余的位置使用null填充
String[] c = l.toArray(new String[0]);
```

除此之外，还有一些有用的辅助方法，用于处理 Java 数组。为了完整性，这里也会介绍。

java.lang.System 类定义了一个 arraycopy() 方法，作用是把一个数组中的指定元素复制到另一个数组的指定位置。这两个数组的类型必须相同，甚至可以是同一个数组：

```
char[] text = "Now is the time".toCharArray();
```

```
char[] copy = new char[100];
// 从text的第4个元素开始,复制10个字符到copy中
// 这10个字符的位置从copy[0]开始
System.arraycopy(text, 4, copy, 0, 10);

// 把一些元素向后移,留出位置插入其他元素
System.arraycopy(copy, 3, copy, 6, 7);
```

Arrays 类还定义了一些有用的静态方法：

```
int[] intarray = new int[] { 10, 5, 7, -3 }; // 由整数组成的数组
Arrays.sort(intarray);                       // 原地排序数组
// 在索引2找到值7
int pos = Arrays.binarySearch(intarray, 7);
// 未找到:返回负数
pos = Arrays.binarySearch(intarray, 12);

// 由对象组成的数组也能排序和搜索
String[] strarray = new String[] { "now", "is", "the", "time" };
Arrays.sort(strarray);  // 排序的结果:{ "is", "now", "the", "time" }

// Arrays.equals()方法比较两个数组中的所有元素
String[] clone = (String[]) strarray.clone();
boolean b1 = Arrays.equals(strarray, clone); // 是的,两个数组相等

// Arrays.fill()方法用于初始化数组的元素
// 一个空数组,所有元素都是0
byte[] data = new byte[100];
// 把元素都设为-1
Arrays.fill(data, (byte) -1);
// 把第5-9个元素设为-2
Arrays.fill(data, 5, 10, (byte) -2);
```

在 Java 中，数组可以视作对象，也可以按照对象的方法处理。假如有个对象 o，可以使用类似下面的代码判断这个对象是否为数组。如果是，则判断是什么类型的数组：

```
Class type = o.getClass();
if (type.isArray()) {
  Class elementType = type.getComponentType();
}
```

8.2 在Java集合框架中使用lambda表达式

Java 8 引入 lambda 表达式的一个主要原因是大幅修改集合 API，让 Java 开发者使用更现代化的编程风格。在 Java 8 发布之前，使用 Java 处理数据结构的方式有点过时。现在，很多语言都支持把集合看成一个整体，而不用打散后再迭代。

事实上，很多 Java 开发者已经使用了替代的数据结构库，获取他们认为集合 API 缺乏的表现力和生产力。升级集合 API 的关键是引入参数能使用 lambda 表达式的新方法，定义需要做什么，而不用管具体怎么做。

有默认方法这个新语言特性的支持，才能在现有的接口中添加新方法（详情参见 4.1.6 节）。没有这个新机制的话，集合接口的原有实现在 Java 8 中不能编译，而且在 Java 8 的运行时中加载时无法链接。

本节简要介绍如何在 Java 集合框架中使用 lambda 表达式。完整的说明参阅 Richard Warburton 写的《Java 8 函数式编程》一书（O'Reilly 出版，http://shop.oreilly.com/product/0636920030713.do）。

8.2.1　函数式方式

Java 8 想实现的方式源于函数式编程语言和风格。我们在 4.5.2 节已经介绍过一些关键的模式，这里会再次介绍，并举些例子。

1. 过滤器

这个模式把集合中的每个元素代入一段代码（返回 `true` 或 `false`），然后使用"通过测试"（即代入元素的那段代码返回 `true`）的元素构建一个新集合。

例如，下面这段代码处理一个由猫科动物名组成的集合，选出是老虎的元素：

```
String[] input = {"tiger", "cat", "TIGER", "Tiger", "leopard"};
List<String> cats = Arrays.asList(input);
String search = "tiger";
String tigers = cats.stream()
                    .filter(s -> s.equalsIgnoreCase(search))
                    .collect(Collectors.joining(", "));
System.out.println(tigers);
```

上述代码的关键是对 `filter()` 方法的调用。`filter()` 方法的参数是一个 lambda 表达式，这个 lambda 表达式接受一个字符串参数，返回布尔值。整个 `cats` 集合中的元素都会代入这个表达式，然后创建一个新集合，只包含老虎（不过有些使用大写）。

`filter()` 方法的参数是一个 `Predicate` 接口的实例。`Predicate` 接口在新包 `java.util.function` 中定义。这是个函数式接口，只有一个非默认方法，因此特别适合 lambda 表达式。

注意，最后还调用了 `collect()` 方法。这个方法是流 API 的重要部分，作用是在 lambda 表达式执行完毕后"收集"结果。下一节会深入介绍这个方法。

`Predicate` 接口有一些十分有用的默认方法，例如用于合并判断条件的逻辑操作方法。假如想把豹子纳入老虎种群，可以使用 `or()` 方法：

```
Predicate<String> p = s -> s.equalsIgnoreCase(search);
Predicate<String> combined = p.or(s -> s.equals("leopard"));
```

```
String pride = cats.stream()
                   .filter(combined)
                   .collect(Collectors.joining(", "));
System.out.println(pride);
```

注意，必须显式创建 `Predicate<String>` 类型的对象 `p`，这样才能在 `p` 上调用默认方法 `or()`，并把另一个 lambda 表达式（也会自动转换成 `Predicate<String>` 类型的对象）传给 `or()` 方法。

2. 映射

Java 8 中的映射模式使用 `java.util.function` 包中的新接口 `Function<T, R>`。这个接口和 `Predicate<T>` 接口一样，是函数式接口，因此只有一个非默认方法——`apply()`。映射模式把一种类型元素组成的集合转换成另一种类型元素组成的集合。这一点在 API 中就体现出来了，因为 `Function<T, R>` 接口有两个不同的类型参数，其中，类型参数 R 的名称表示这个方法的返回类型。

下面看一个使用 `map()` 方法的示例代码：

```
List<Integer> namesLength = cats.stream()
                .map(String::length)
                .collect(Collectors.toList());
System.out.println(namesLength);
```

3. 遍历

映射和过滤器模式的作用是以一个集合为基础，创建另一个集合。在完全支持函数式编程的语言中，除了这种方式之外，还需要 lambda 表达式的主体处理各个元素时不影响原来的集合。用计算机科学的术语来说，这意味着 lambda 表达式的主体“不能有副作用”。

当然，在 Java 中经常需要处理可变的数据，所以新的集合 API 提供了一个方法，在遍历集合时修改元素——`forEach()` 方法。这个方法的参数是一个 `Consumer<T>` 类型的对象。`Consumer<T>` 是函数式接口，要求使用副作用执行操作（然而，到底会不会真得修改数据不是那么重要）。因此，能转换成 `Consumer<T>` 类型的 lambda 表达式，其签名为 `(T t) -> void`。下面是一个使用 `forEach()` 方法的简单示例：

```
List<String> pets =
  Arrays.asList("dog", "cat", "fish", "iguana", "ferret");
pets.stream().forEach(System.out::println);
```

在这个示例中，我们只是把集合中的每个元素打印出来。不过，我们把 lambda 表达式写成了一种特殊的方法引用。这种方法引用叫*受限的方法引用*（bound method reference），因为需要指定对象（这里指定的对象是 `System.out`，`System` 类的公开静态字段）。这个方法引用和下面的 lambda 表达式等效：

```
s -> System.out.println(s);
```

当然，根据方法的签名，这样写能明确表明 lambda 表达式要转换成一个实现 Consumer<? super String> 接口类型的实例。

> 不是说 map() 或 filter() 方法一定不能修改元素。不要使用这两个方法修改元素，这只是一种约定，每个 Java 程序员都要遵守。

在结束本节之前，还有最后一个函数式技术要介绍。这种技术把集合中的元素聚合成一个值，详情参见下一小节。

4. 化简

下面介绍 reduce() 方法。这个方法实现的是化简模式，包含一系列相关的类似运算，有时也称为合拢或聚合运算。

在 Java 8 中，reduce() 方法有两个参数：一个是初始值，一般叫作单位值（或零值）；另一个参数是一个函数，逐步执行。这个函数属于 BinaryOperator<T> 类型。BinaryOperator<T> 也是函数式接口，有两个类型相同的参数，返回值也是同一类型。reduce() 方法的第二个参数是一个 lambda 表达式，接受两个参数。在 Java 的文档中，reduce() 方法的签名是：

```
T reduce(T identity, BinaryOperator<T> aggregator);
```

reduce() 方法的第二个参数可以简单地理解成，在处理流的过程中“累积计数”：首先合并单位值和流中的第一个元素，得到第一个结果，然后再合并这个结果和流中的第二个元素，以此类推。

把 reduce() 方法的实现设想成下面这样有助于理解其作用：

```
public T reduce(T identity, BinaryOperator<T> aggregator) {
    T runningTotal = identity;
    for (T element : myStream) {
        runningTotal = aggregator.apply(runningTotal, element);
    }

    return result;
}
```

> 实际上，reduce() 方法的实现比这复杂得多，如果数据结构和运算有需要，甚至还可以并行执行。

下面看一个使用 reduce() 方法的简单示例，这个示例计算几个质数之和：

```
double sumPrimes = ((double)Stream.of(2, 3, 5, 7, 11, 13, 17, 19, 23)
        .reduce(0, (x, y) -> {return x + y;}));
System.out.println("Sum of some primes: " + sumPrimes);
```

你可能注意到了，本节举的所有示例中，都在 List 实例上调用了 stream() 方法。这是集合 API 演进的一部分——一开始选择这种方式是因为部分 API 有这方面的需求，但后来证实，这是极好的抽象。下面详细讨论流 API。

8.2.2 流API

库的设计者之所以引入流 API，是因为集合核心接口的大量实现已经广泛使用。这些实现在 Java 8 和 lambda 表达式之前就已存在，因此没有执行任何函数式运算的方法。更糟的是，map() 和 filter() 等方法从未出现在集合 API 的接口中，实现这些接口的类可能已经使用这些名称定义了方法。

为了解决这个问题，设计者引入了一层新的抽象——Stream。Stream 对象可以通过 stream() 方法从集合对象上生成。设计者引入这个全新的 Stream 对象是为了避免方法名冲突，这的确在一定程度上减少了冲突的几率，因为只有包含 stream() 方法的实现才会受到影响。

在处理集合的新方式中，Stream 对象的作用和 Iterator 对象类似。总体思想是让开发者把一系列操作（也叫“管道”，例如映射、过滤器或化简）当成一个整体运用在集合上。具体执行的各个操作一般使用 lambda 表达式表示。

在管道的末尾需要收集结果，或者再次“具化”为真正的集合。这一步使用 Collector 对象完成，或者以“终结方法”（例如 reduce()）结束管道，返回一个具体的值，而不是另一个流。总的来说，处理集合的新方式类似下面这样：

```
         stream()   filter()   map()   collect()
Collection -> Stream -> Stream -> Stream -> Collection
```

Stream 类相当于一系列元素，一次访问一个元素（不过有些类型的流也支持并行访问，可以使用多线程方式处理大型集合）。Stream 对象和 Iterator 对象的工作方式一样，依次读取每个元素。

和 Java 中的大多数泛型类一样，Stream 类也使用引用类型参数化。不过，多数情况下，其实需要使用基本类型，尤其是 int 和 double 类型构成的流，但是又没有 Stream<int> 类型，所以 java.util.stream 包提供了专用的（非泛型）类，例如 IntStream 和 DoubleStream。这些类是 Stream 类的基本类型特化，其 API 和一般的 Stream 类十分类似，不过在适当的情况下会使用基本类型的值。

例如，在前面 reduce() 方法的示例中，多数时候，在管道中使用的其实就是 Stream 类的基本类型特化。

1. 惰性求值

其实，流比迭代器（甚至是集合）通用，因为流不用管理数据的存储空间。在早期的 Java 版本中，总是假定集合中的所有元素都存在（一般存储在内存中），不过有些处理方式也能避开这个问题，例如坚持在所有地方都使用迭代器，或者让迭代器即时构建元素。可是，这些方式既不十分便利，也不那么常用。

然而，流是管理数据的一种抽象，不关心存储细节。因此，除了有限的集合之外，流还能处理更复杂的数据结构。例如，使用 Stream 接口可以轻易实现无限流，处理一切平方数。实现方式如下所示：

```
public class SquareGenerator implements IntSupplier {
    private int current = 1;

    @Override
    public synchronized int getAsInt() {
        int thisResult = current * current;
        current++;
        return thisResult;
    }
}

IntStream squares = IntStream.generate(new SquareGenerator());
PrimitiveIterator.OfInt stepThrough = squares.iterator();
for (int i = 0; i < 10; i++) {
    System.out.println(stepThrough.nextInt());
}
System.out.println("First iterator done...");

// 只要想就可以一直这样进行下去……
for (int i = 0; i < 10; i++) {
    System.out.println(stepThrough.nextInt());
}
```

通过构建上述无限流，我们能得出一个重要结论：不能使用 collect() 这样的方法。这是因为无法把整个流具化为一个集合（在创建所需的无限个对象之前就会耗尽内存）。因此，我们采取的方式必须在需要时才从流中取出元素。其实，我们需要的是按需读取下一个元素的代码。为了实现这种操作，需要使用一个关键技术——*惰性求值*（lazy evaluation）。这个技术的本质是，需要时才计算值。

> 惰性求值对 Java 来说是个重大的变化，在 JDK 8 之前，表达式赋值给变量（或传入方法）后会立即计算它的值。这种立即计算值的方式我们已经熟知，术语叫“及早求值”（eager evaluation）。在多数主流编程语言中，“及早求值”都是计算表达式的默认方式。

幸好，实现惰性求值的重担几乎都落在了库的编写者身上，开发者则轻松得多，而且使用流 API 时，大多数情况下 Java 开发者都无需仔细考虑惰性求值。下面以一个示例结束对流的讨论。这个示例使用 reduce() 方法计算几个莎士比亚语录的平均单词长度：

```
String[] billyQuotes = {"For Brutus is an honourable man",
  "Give me your hands if we be friends and Robin shall restore amends",
  "Misery acquaints a man with strange bedfellows"};
List<String> quotes = Arrays.asList(billyQuotes);

// 创建一个临时集合,保存单词
List<String> words = quotes.stream()
        .flatMap(line -> Stream.of(line.split(" +")))
        .collect(Collectors.toList());
long wordCount = words.size();

// 校正为double类型只是为了避免Java按照整数方式计算除法
double aveLength = ((double) words.stream()
        .map(String::length)
        .reduce(0, (x, y) -> {return x + y;})) / wordCount;
System.out.println("Average word length: " + aveLength);
```

这个示例用到了 flatMap() 方法。在这个示例中，向 flatMap() 方法传入一个字符串 line，得到的是一个由字符串组成的流，流中的数据是拆分一句话得到的所有单词。然后再“整平”这些单词，把处理每句话得到的流都合并到一个流中。

这样做的目的是把每句话都拆分成单个单词，然后再组成一个总流。为了计算单词数量，我们创建了一个对象 words。其实，在管道处理流的过程中会“中断”，再次具化，把单词存入集合，在流操作恢复之前获取单词的数量。

这一步完成之后，下一步是化简运算，先计算所有语录中的单词总长度，然后再除以已经获取的单词数量。记住，流是惰性抽象，如果要执行及早操作（例如，计算流下面的集合大小），得重新具化为集合。

2. 处理流的实用默认方法

借着引入流 API 的机会，Java 8 向集合库引入了一些新方法。现在 Java 已经支持默认方法，因此可以向集合接口中添加新方法，而不会破坏向后兼容性。

新添加的方法中有一些是“基架方法”，用于创建抽象的流，例如 Collection::stream、Collection::parallelStream 和 Collection::spliterator（这个方法可以细分为 List::spliterator 和 Set::spliterator）。

另一些则是“缺失方法”，例如 Map::remove 和 Map::replace。List::sort 也属于“缺失方法”，在 List 接口中的定义如下所示：

```
// 其实是把具体操作交给Collections类的辅助方法完成
public default void sort(Comparator<? super E> c) {
```

```
        Collections.<E>sort(this, c);
    }
```

Map::putIfAbsent 也是缺失方法，根据 java.util.concurrent 包中 ConcurrentMap 接口的同名方法改写。

另一个值得关注的缺失方法是 Map::getOrDefault，程序员使用这个方法能省去很多检查 null 值的繁琐操作，因为如果找不到要查询的键，这个方法会返回指定的值。

其余的方法则使用 java.util.function 接口提供额外的函数式技术。

- Collection::removeIf

  这个方法的参数是一个 Predicate 对象，它会迭代整个集合，把满足判断条件的元素移除。

- Map::forEach

  这个方法只有一个参数，是一个 lambda 表达式；而这个 lambda 表达式有两个参数（一个是键的类型，一个是值的类型），返回 void。这个 lambda 表达式会转换成 BiConsumer 对象，应用在映射中的每个键值对上。

- Map::computeIfAbsent

  这个方法有两个参数：键和 lambda 表达式。lambda 表达式的作用是把键映射到值上。如果映射中没有指定的键（第一个参数），那就使用 lambda 表达式计算一个默认值，然后存入映射。

（其他值得学习的方法：Map::computeIfPresent、Map::compute 和 Map::merge。）

8.3　小结

本章介绍了 Java 集合库，也说明了如何开始使用 Java 实现的基本和经典数据结构。我们学习了通用的 Collection 接口，以及 List、Set 和 Map 接口；学习了处理集合的原始迭代方式，也介绍了 Java 8 从函数式编程语言借鉴来的新方式。最后，我们学习了流 API，发现这种新方式更通用，而且处理复杂的编程概念时比经典方式更具表现力。

我们继续学习。下一章继续讨论数据，会介绍一些常见任务的处理方式，例如处理文本和数字数据，还会介绍 Java 8 引入的新日期和时间库。

第 9 章

处理常见的数据格式

编程的多数任务是处理不同格式的数据。本章介绍 Java 处理两大类数据的方式——文本和数字。后半部分则集中介绍处理日期和时间的方式。这一部分特别有趣，因为 Java 8 提供了处理日期和时间的全新 API。我们会先稍微深入地介绍新接口，然后再简要讨论以前的日期和时间 API。

很多应用仍在使用以前的 API，所以开发者需要知道旧的处理方式。不过，新 API 太好用了，建议尽早转用。在讨论这些复杂的格式之前，先来说说文本数据和字符串。

9.1 文本

我们已经在很多场合见过 Java 的字符串。字符串由一系列 Unicode 字符组成，是 `String` 类的实例。字符串是 Java 程序最常处理的数据类型之一（可以使用第 13 章介绍的 `jmap` 工具证实这一点）。

本节会深入介绍 `String` 类，并弄清为什么字符串在 Java 语言中占据如此重要的地位。本节末尾还会介绍正则表达式，这是十分常用的抽象方式，用于搜索文本中的模式匹配（也是程序员的传统工具）。

9.1.1 字符串的特殊句法

Java 语言使用某种特殊的方式处理 `String` 类。虽然字符串不是基本类型，但十分常用，所以 Java 的设计者觉得有必要提供一些特殊的句法特性，便于处理字符串。下面通过一些示例介绍 Java 为字符串提供的特殊句法特性。

1. 字符串字面量

第 2 章介绍过，Java 允许把一系列字符放在双引号中创建字面量字符串对象。例如：

```
String pet = "Cat";
```

如果没有这种特殊的句法，就要编写大量不友好的代码，例如：

```
char[] pullingTeeth = {'C', 'a', 't'};
String pet = new String(pullingTeeth);
```

这样很快就会把代码变得冗长乏味，因此，Java 像所有现代编程语言一样，提供了简单的字符串字面量句法。字符串字面量是完全有效的对象，所以类似下面这种代码是完全合法的：

```
System.out.println("Dog".length());
```

2. toString()方法

这个方法在 Object 类中定义，作用是方便把任何对象转换成字符串。有了这个方法，就可以使用 System.out.println() 方法轻易打印任何对象。System.out.println() 方法其实是 PrintStream::println，因为 System.out 是 PrintStream 类型的静态字段。我们来看一下这个方法是如何定义的：

```
public void println(Object x) {
    String s = String.valueOf(x);
    synchronized (this) {
        print(s);
        newLine();
    }
}
```

这个方法使用静态方法 String::valueOf() 创建了一个新字符串：

```
public static String valueOf(Object obj) {
    return (obj == null) ? "null" : obj.toString();
}
```

> println() 方法没有直接使用 toString() 方法，而使用了静态方法 valueOf()，这么做是为了避免 obj 为 null 时抛出 NullPointerException 异常。

这种定义方式让任何对象都能调用 toString() 方法，也十分有利于 Java 提供的另一种重要的句法特性——字符串连接。

3. 字符串连接

在 Java 中，可以把一个字符串“添加”到另一个字符串的末尾，创建新字符串——这是

一个语言特性，叫作*字符串连接*，使用运算符 + 实现。连接字符串时，先创建一个使用 StringBuilder 对象表示的“工作区”，其内容和原始字符串中的字符序列一样。

然后更新 StringBuilder 对象，把另一个字符串中的字符添加到末尾。最后，在 StringBuilder 对象（现在这个对象包含两个字符串中的字符）上调用 toString() 方法，得到一个包含所有字符的新字符串。使用 + 运算符连接字符串时，javac 会自动创建上述所有代码。

连接后得到的是全新的 String 对象，这一点从下面的示例可以看出：

```
String s1 = "AB";
String s2 = "CD";

String s3 = s1;
System.out.println(s1 == s3); // 是不是同一个对象？

s3 = s1 + s2;
System.out.println(s1 == s3); // 还是不是同一个对象？
System.out.println(s1);
System.out.println(s3);
```

这个连接字符串的示例直接表明，+ 运算符没有就地修改（或改变）s1。这个示例也体现了一个通用规则：Java 的字符串是不可变的。也就是说，选定组成字符串的字符并创建 String 对象后，字符串的内容就不能改变了。这是 Java 语言的一个重要规则，下面稍微深入地讨论一下。

9.1.2　字符串的不可变性

为了“修改”字符串，就像前面连接字符串那样，其实需要创建一个过渡的 StringBuilder 对象作为暂存区，然后在这个对象上调用 toString() 方法，创建一个新 String 实例。下面通过代码演示这个过程：

```
String pet = "Cat";
StringBuilder sb = new StringBuilder(pet);
sb.append("amaran");
String boat = sb.toString();
System.out.println(boat);
```

如果编写的是下述代码，javac 就会生成类似上面的代码：

```
String pet = "Cat";
String boat = pet + "amaran";
System.out.println(boat);
```

当然，除了能由 javac 隐式使用之外，如前所示，也可以直接使用 StringBuilder 类。

除了 StringBuilder 类，Java 还有 StringBuffer 类。StringBuffer 类在最早的 Java 版本中出现，新编写的程序不要使用这个类——应该使用 StringBuilder 类，除非确实需要在多个线程之间共享构建的新字符串。

字符串的不可变性是极其有用的语言特性。假如 + 运算符直接修改字符串，而不是创建新字符串，那么，只要某个线程连接了两个字符串，其他所有线程都能看到这个变化。对大多数程序来说，这种行为没什么用，所以不可变性更合理。

哈希码和事实不可变性

第 5 章说明方法必须满足的契约（contract）[1] 时，见过 hashCode() 方法。我们来看一下 String::hashCode() 方法在 JDK 源码中是怎么定义的：

```
public int hashCode() {
    int h = hash;
    if (h == 0 && value.length > 0) {
        char val[] = value;

        for (int i = 0; i < value.length; i++) {
            h = 31 * h + val[i];
        }
        hash = h;
    }
    return h;
}
```

hash 字段保存的是字符串的哈希码，value 字段是 char[] 类型，保存的是组成字符串的字符。从上述代码可以看出，计算哈希码时会遍历字符串中的所有字符。因此，所执行的机器指令数量和字符串中的字符数量成正比。对超大型字符串来说，要花点时间才能算出哈希码。不过，Java 不会预先计算好哈希码，只在需要使用时才计算。

运行这个方法时，会迭代数组中的字符，算出哈希码。迭代结束后，退出 for 循环，把算出的哈希码存入 hash 字段。如果再次调用这个方法，因为已经算出了哈希码，所以会直接使用缓存的值。因此，后续再调用 hashCode() 方法，会立即返回

计算字符串哈希码的过程是*良性数据竞争*的一例。运行在多线程环境中的程序，多个线程可能会竞相计算哈希码。不过，这些线程最终会得到完全相同的结果，因此才说这种竞争是“良性的”。

String 类的字段，除了 hash 之外都声明为 final。所以，严格来说，Java 的字符串并不是不可变的。不过，hash 字段缓存的值是根据其他字段计算而来的，而这些字段的值都是不

注 1：契约指方法的行为所符合的特定标准。——译者注

可变的，因此，只要选定了字符串的内容，那么表现出来的行为就像是不可变的一样。具有这种特性的类称为*事实不可变的类*——现实中很少见到这种类，程序员往往可以忽略真正不可变的数据和事实不可变的数据之间的区别。

9.1.3　正则表达式

Java 支持*正则表达式*（regular expression，经常简称 regex 或 regexp）。正则表达式表示的是用于扫描和匹配文本的搜索模式。一个正则表达式就是我们想搜索的字符序列。有些正则表达式很简单，例如 `abc`，这个正则表达式的意思是，在要搜索的文本中查找连在一起的“abc”。注意，搜索模式匹配的文本可以出现零次、一次或多次。

最简单的正则表达式只包含字符字面量序列，例如 `abc`。不过，正则表达式使用的语言能表达比字面量序列复杂和精细的模式。例如，正则表达式能表示匹配下述内容的模式：

- 一个数字
- 任何字母
- 任意个字母，但字母只能在 a 到 j 之间，大小写不限
- a 和 b 之间有任意四个字符

编写正则表达式的句法虽然简单，但是因为可能要编写复杂的模式，所以往往写出的正则表达式不能实现真正想要的模式。因此，使用正则表达式时，一定要充分测试，既要有能通过的测试用例，也要有失败的测试用例。

为了表示复杂的模式，在正则表达式中要使用*元字符*。这种特殊的字符要特别对待。元字符的作用类似于 Unix 或 Windows shell 中使用的 `*` 字符。在 shell 中，我们知道 `*` 字符不能按照字面量理解，而是表示“任意字符”。在 Unix 中，如果想列出当前目录中的全部 Java 源码文件，可以执行下述命令：

```
ls *.java
```

正则表达式中的元字符和 `*` 字符的作用类似，和 shell 中可以使用的特殊字符相比，元字符数量更多，用起来也更灵活，而且有不同的意义，所以不要混淆了。

我们看个例子。假如我们要编写一个拼写检查程序，但不严格限制只能使用英式英语或美式英语的拼写方式。也就是说，honor 和 honour 都是有效的拼写。这个要求通过正则表达式很容易实现。

Java 使用 `Pattern` 类（在 `java.util.regex` 包中）表示正则表达式。不过，这个类不能直接实例化，只能使用静态工厂方法 `compile()` 创建实例。然后，再从模式上创建某个输入字符串的 `Matcher` 对象，用于匹配输入字符串。例如，我们来研究一下莎士比亚写的戏剧《裘里斯 · 凯撒》：

```
Pattern p = Pattern.compile("honou?r");

String caesarUK = "For Brutus is an honourable man";
Matcher mUK = p.matcher(caesarUK);

String caesarUS = "For Brutus is an honorable man";
Matcher mUS = p.matcher(caesarUS);

System.out.println("Matches UK spelling? " + mUK.find());
System.out.println("Matches US spelling? " + mUS.find());
```

使用 Matcher 类时要小心，因为它有个名为 matches() 的方法。这个方法判断模式是否匹配整个输入字符串。如果模式只从字符串的中间开始匹配，这个方法会返回 false。

上述示例第一次用到了正则表达式元字符——honou?r 模式中的 ?。这个元字符的意思是，“前一个字符是可选的”，所以这个模式既能匹配 honour 也能匹配 honor。下面再看个例子。假设我们既想匹配 minimize，又想匹配 minimise（这种拼写在英式英语中较常见），那么，可以使用方括号表示能匹配一个集中的任意一个字符（只能是一个），如下所示：

```
Pattern p = Pattern.compile("minimi[sz]e");
```

表 9-1 列出了 Java 正则表达式可以使用的一些元字符。

表9-1：正则表达式元字符

元字符	意　　义	备　　注
?	可选字符——出现零次或一次	
*	前一个字符出现零次或多次	
+	前一个字符出现一次或多次	
{M,N}	前一个字符出现 M 到 N 次	
\d	一个数字	
\D	一个不是数字的字符	
\w	一个组成单词的字符	数字、字母和 _
\W	一个不能组成单词的字符	
\s	一个空白字符	
\S	一个不是空白的字符	
\n	换行符	
\t	制表符	
.	任意一个字符	在 Java 中不包含换行符
[ ]	方括号中的任意一个字符	叫作字符组
[^]	不在方括号中的任意一个字符	叫作排除字符组
( )	构成一组模式元素	叫作组（或捕获组）

（续）

元字符	意　　义	备　　注
|	定义可选值	实现逻辑或
^	字符串的开头	
$	字符串的末尾	

除此之外还有一些，不过这些是基本的元字符。使用这些元字符可以编写更复杂的正则表达式，匹配本节前面给出的示例：

```
// 注意,必须使用\\,因为我们需要的是字面量\,而Java使用单个\转义字符
String pStr = "\\d"; // 一个数字
String text = "Apollo 13";
Pattern p = Pattern.compile(pStr);
Matcher m = p.matcher(text);
System.out.print(pStr + " matches " + text + "? " + m.find());
System.out.println(" ; match: " + m.group());


pStr = "[a..zA..Z]"; // 任意一个字母
p = Pattern.compile(pStr);
m = p.matcher(text);
System.out.print(pStr + " matches " + text + "? " + m.find());
System.out.println(" ; match: " + m.group());

// 任意个字母,但字母只能在a到j之间,大小写不限
pStr = "([a..jA..J]*)";
p = Pattern.compile(pStr);
m = p.matcher(text);
System.out.print(pStr + " matches " + text + "? " + m.find());
System.out.println(" ; match: " + m.group());

text = "abacab";
pStr = "a....b"; // a和b之间有四个字符
p = Pattern.compile(pStr);
m = p.matcher(text);
System.out.print(pStr + " matches " + text + "? " + m.find());
System.out.println(" ; match: " + m.group());
```

本节结束之前，我们还要介绍 Java 8 添加到 `Pattern` 类中的一个新方法：`asPredicate()`。引入这个方法的目的是，让开发者通过简单的方式把正则表达式与 Java 集合和对 lambda 表达式的支持联系起来。

假如有一个正则表达式和一个由字符串组成的集合，很自然地会出现这个问题：“哪些字符串匹配这个正则表达式？”为了回答这个问题，我们可以使用过滤器模式，并使用辅助方法 `asPredicate()` 把正则表达式转换成 `Predicate` 对象，如下所示：

```
String pStr = "\\d"; // 一个数字
Pattern p = Pattern.compile(pStr);
```

```
String[] inputs = {"Cat", "Dog", "Ice-9", "99 Luftballoons"};
List<String> ls = Arrays.asList(inputs);
List<String> containDigits = ls.stream()
                               .filter(p.asPredicate())
                               .collect(Collectors.toList());
System.out.println(containDigits);
```

Java 对文本处理的原生支持完全能胜任大多数商业应用对文本处理任务的一般要求。更高级的任务，例如搜索并处理超大型数据集，或复杂的解析操作（包括形式语法），超出了本书范畴，不过要知道，Java 的生态系统很庞大，有很多有用的库，而且有很多用于文本处理和分析的专用技术。

9.2 数字和数学运算

本节深入讨论 Java 对数字类型的支持。具体来说，我们会讨论 Java 使用二进制补码表示的整数类型，还会介绍浮点数的表示方式，以及由此引起的一些问题。最后，还会举些例子，说明如何使用 Java 库提供的函数做标准的数学运算。

9.2.1 Java表示整数类型的方式

在 2.3 节说过，Java 的整数类型都带符号，也就是说，所有整数类型既可以表示正数，也可以表示负数。因为计算机只能处理二进制数，所以唯一合理的方式是把所有可能的位组合分成两半，使用其中一半表示负数。

我们以 Java 的 byte 类型为例，说明 Java 是如何表示整数的。byte 类型的数字占 8 位，因此能表示 256 个不同的数字（即 128 个正数和 128 个负数）。0b0000_0000 表示的是零（记得吗，在 Java 中可以使用 0b<binary digits> 这样的句法表示二进制数），因此很容易看出表示正数的位组合：

```
byte b = 0b0000_0001;
System.out.println(b); // 1

b = 0b0000_0010;
System.out.println(b); // 2

b = 0b0000_0011;
System.out.println(b); // 3

// ...

b = 0b0111_1111;
System.out.println(b); // 127
```

如果设定了 byte 类型数字的第一位，符号应该改变（因为表示正数已经用完了其他位）。所以，0b1000_0000 组合应该表示某个负数，不过是哪个负数呢？

> 按照我们定义事物的方式，在这种表示方式中，很容易找到一种识别位组合是否表示负数的方式：如果位组合的高位是 1，表示的就是负数。

我们来分析一下所有位都为 1 的位组合：0b1111_1111。如果在这个数字上加 1，那么得到的结果就会超出存储 byte 类型所需的 8 位，变成 0b1_0000_0000。如果我们强制把这个数变成 byte 类型，就要忽略超出的那一位，变成 0b0000_0000，也就是零。因此，顺其自然，我们把所有位都为 1 的位组合认定为 -1。这样也就得到了符合常理的算术规则，如下所示：

```
b = (byte) 0b1111_1111; // -1
System.out.println(b);
b++;
System.out.println(b);

b = (byte) 0b1111_1110; // -2
System.out.println(b);
b++;
System.out.println(b);
```

最后，我们来看一下 0b1000_0000 表示的是什么数字。这个位组合表示的是这个类型能表示的最小负数，所以对 byte 类型来说：

```
b = (byte) 0b1000_0000;
System.out.println(b); // -128
```

这种表示方式叫*二进制补码*，带符号的正数最常使用这种表示方式。为了有效使用，只需记住两点：

- 所有位都为 1 的位组合表示 −1；
- 如果设定了高位，表示的是负数。

Java 的其他整数类型（short、int 和 long）和 byte 的行为十分相似，只不过位数更多。但是 char 类型有所不同，因为它表示的是 Unicode 字符，不过可以使用某种方式表示成无符号的 16 位数字类型。Java 程序员一般不会把 char 当成整数类型。

9.2.2　Java中的浮点数

计算机使用二进制表示数字。我们已经说明了 Java 如何使用二进制补码表示整数，那么分数或小数怎么办？和几乎所有现代编程语言一样，Java 使用*浮点运算*表示分数和小数。下面说明具体的实现方式，先说十进制（常规小数），再说二进制。Java 在 java.lang.Math 类中定义了两个最重要的数学常数，e 和 π，如下所示：

```
public static final double E = 2.7182818284590452354;
public static final double PI = 3.14159265358979323846;
```

当然，这两个常数其实是*无理数*，无法通过分数或任何有限的小数精确表示。[2] 也就是说，在计算机中表示时，无论如何想尽办法，都有舍入误差。假如我们只想使用 π 的前八位数，而且想把这些数字表示成一个整数，那么可以像这样表示：

$$314159265 \times 10^{-8}$$

这种表示方式显露了实现浮点数的基本方式。我们使用其中几位表示数字的有效位（在这个例子中是 `314159265`），另外几位则表示底数的*指数*（在这个例子中是 -8）。有效位表示的是*有效数字*，而指数说明为了得到所需的数字，要对有效数字做上移操作还是下移操作。

当然，目前举的例子都使用十进制。可是计算机使用二进制，所以我们要以二进制为例说明浮点数的实现方式，由此也增加了一些复杂度。

> `0.1` 这个数不能使用有限的二进制数位表示。也就是说，人类关心的所有计算，使用浮点数表示时几乎都会失去精度，而且根本无法避免舍入误差。

下面举个例子，说明舍入问题：

```
double d = 0.3;
System.out.println(d); // 精心挑选的,避免显示难看的表示

double d2 = 0.2;
// 应该是-0.1,但打印出来的是-0.09999999999999998
System.out.println(d2 - d);
```

规范浮点计算的标准是 IEEE-754，Java 就是基于这个标准实现的浮点数。按照这个标准，表示标准精度的浮点数要使用 24 个二进制数，表示双精度浮点数要使用 53 个二进制数。

第 2 章简略提到过，如果有硬件特性支持，Java 能表示比标准要求的更精确的浮点数。如果需要完全兼容其他平台（可能是较旧的平台），可以使用 `strictfp` 关键字禁用这种行为，强制遵守 IEEE-754 标准。但这种情况极其少见，几乎不用这么做，绝大多数程序员都不会用到（甚至见不到）这个关键字。

`BigDecimal`类

程序员使用浮点数时，舍入误差始终是个让人头疼的问题。为了解决这个问题，Java 提供

注 2：其实，这是两个已知的*超越数*。

了 java.math.BigDecimal 类，这个类以小数形式实现任意精度的计算。使用这个类可以解决有限个二进制位无法表示 0.1 的问题，不过在 BigDecimal 对象和 Java 基本类型之间相互转换时还是会遇到一些边缘情况，如下所示：

```
double d = 0.3;
System.out.println(d);

BigDecimal bd = new BigDecimal(d);
System.out.println(bd);

bd = new BigDecimal("0.3");
System.out.println(bd);
```

可是，就算所有计算都按照十进制进行，仍然有些数，例如 1/3，无法使用有尽小数表示。我们来看一下尝试使用 BigDecimal 表示这种数时会出现什么状况：

```
bd = new BigDecimal(BigInteger.ONE);
bd.divide(new BigDecimal(3.0));
System.out.println(bd); // 应该是1/3
```

因为 BigDecimal 无法精确表示 1/3，所以调用 divide() 方法时会抛出 ArithmeticException 异常。因此，使用 BigDecimal 时一定要清醒地意识到哪些运算的结果可能是无尽小数。更糟的是，ArithmeticException 是运行时的未检异常，所以出现这种异常时，Java 编译器根本不会发出警告。

最后，关于浮点数，所有高级程序员都应该阅读 David Goldberg 写的“What Every Computer Scientist Should Know About Floating-Point Arithmetic”。网上可以轻易找到这篇文章，而且可以免费阅读。[3]

9.2.3 Java的数学函数标准库

在结束 Java 对数字数据和数学运算的介绍之前，我们还要简单说明 Java 标准库中的一些函数。这些函数基本上都是静态辅助方法，在 java.lang.Math 类中定义，包括如下。

- abs()

  返回指定数的绝对值。不同的基本类型都有对应的重载方法。

- 三角函数

  计算正弦、余弦和正切等的基本函数。Java 还提供了双曲线版本和反函数（例如反正弦）。

注 3：这篇论文发表在 Computing Surveys 杂志 1991 年 3 月号上。Sun 公司的《数值计算指南》手册里有中文版，地址：http://docs.oracle.com/cd/E19059-01/stud.9/817-7888/817-7888.pdf。——译者注

- `max()`和`min()`

  这两个是重载的函数，分别返回两个参数（属于同一种数字类型）中较大的数和较小的数。

- `floor()`

  返回比指定参数（`double` 类型）小的最大整数。`ceil()` 方法返回比指定参数大的最小整数。

- `pow()`、`exp()`和`log()`

  `pow()` 方法以第一个参数为底数，第二个参数为指数，计算次方；`exp()` 方法做指数运算；`log()` 方法做对数运算。`log10()` 方法计算对数时以 10 为底数，而不是自然常数。

下面举一些简单的示例，说明如何使用这些函数：

```
System.out.println(Math.abs(2));
System.out.println(Math.abs(-2));

double cosp3 = Math.cos(0.3);
double sinp3 = Math.sin(0.3);
System.out.println((cosp3 * cosp3 + sinp3 * sinp3)); // 始终为1.0

System.out.println(Math.max(0.3, 0.7));
System.out.println(Math.max(0.3, -0.3));
System.out.println(Math.max(-0.3, -0.7));

System.out.println(Math.min(0.3, 0.7));
System.out.println(Math.min(0.3, -0.3));
System.out.println(Math.min(-0.3, -0.7));

System.out.println(Math.floor(1.3));
System.out.println(Math.ceil(1.3));
System.out.println(Math.floor(7.5));
System.out.println(Math.ceil(7.5));

System.out.println(Math.round(1.3)); // 返回值为long类型
System.out.println(Math.round(7.5)); // 返回值为long类型

System.out.println(Math.pow(2.0, 10.0));
System.out.println(Math.exp(1));
System.out.println(Math.exp(2));
System.out.println(Math.log(2.718281828459045));
System.out.println(Math.log10(100_000));
System.out.println(Math.log10(Integer.MAX_VALUE));

System.out.println(Math.random());
System.out.println("Let's toss a coin: ");
if (Math.random() > 0.5) {
    System.out.println("It's heads");
} else {
    System.out.println("It's tails");
}
```

最后，我们简要讨论一下 Java 的 `random()` 函数。首次调用这个方法时，会创建一个 `java.util.Random` 类新实例。这是个伪随机数生成器（Pseudorandom Number Generator，PRNG）：生成的数看起来是随机的，其实是由一个数学公式生成的。[4]Java 的 PRNG 使用的公式十分简单，例如：

```
// 摘自java.util.Random类
public double nextDouble() {
    return (((long)(next(26)) << 27) + next(27)) * DOUBLE_UNIT;
}
```

如果伪随机数序列总是从同一个地方开始，那么就会生成完全相同的数字流。为了解决这个问题，可以向 PRNG 提供一个尽可能提升随机性的种子值。为了保证种子值的随机性，Java 会使用 CPU 计数器中的值，这个值一般用于高精度计时。

Java 原生的随机数生成机制能满足多数常规应用，但某些专业应用（特别是密码相关的应用和某些模拟应用）的要求严格得多。如果要开发这类应用，请咨询已经在这些领域中的程序员，寻求专业建议。

对文本和数字数据的介绍结束了，下面介绍另一种最常遇到的数据种类：日期和时间。

9.3 在Java 8中处理日期和时间

几乎所有商业应用软件都具有一些日期和时间的概念。建模真实世界的事件或活动时，知道事件什么时候发生可以对后续报告和域对象的比较都很重要。Java 8 完全改变了开发者处理日期和时间的方式。本节介绍 Java 8 引入的新概念。在之前的版本中，只能通过 `java.util.Date` 类处理日期和时间，而且这个类没有建模这些概念。使用旧 API 的代码应该尽早转用新 API。

9.3.1 介绍Java 8的日期和时间API

Java 8 引入了一个新包 `java.time`，包含了多数开发者都会用到的核心类。这个包分为四个子包。

- `java.time.chrono`

  开发者使用的历法不符合 ISO 标准时，需要与之交互的其他纪年法。例如日本历法。

注 4：在计算机中很难生成真正的随机数，也很少有这方面的需求。如果真想生成真正的随机数，一般都需要特殊的硬件支持。

- `java.time.format`

  这个包中的 `DateTimeFormatter` 类用于把日期和时间对象转换成字符串，以及把字符串解析成日期和时间对象。

- `java.time.temporal`

  包含日期和时间核心类所需的接口，还抽象了一些日期方面的高级操作（例如查询和调节器）。

- `java.time.zone`

  底层时区规则使用的类；多数开发者都用不到这个包。

表示时间时，最重要的概念之一是，某个实体时间轴上的瞬时点。既然这个概念在狭义相对论等理论中已经有了完善的定义，那么在计算机中表示时间就要做些假设。Java 8 使用一个 `Instant` 对象表示一个时间点，而且做了下述关键假设：

- 表示的秒数不能超出 `long` 类型的取值范围；
- 表示的时间不能比纳秒还精细。

因此，能表示的时间受到当前计算机系统的能力所限制。不过，还有一个基本概念需要介绍。

`Instant` 对象是时空中的单一事件。可是，程序员经常要处理的却是两个事件之间的时间间隔，所以 Java 8 还引入了 `java.time.Duration` 类。这个类会忽略可能出现的日历效应（例如夏令时）。了解瞬时和事件持续时间的基本概念之后，我们来看一下瞬时的具体表现。

1. 时间戳的组成

图 9-1 展示了使用不同方式分解时间戳得到的各个部分。

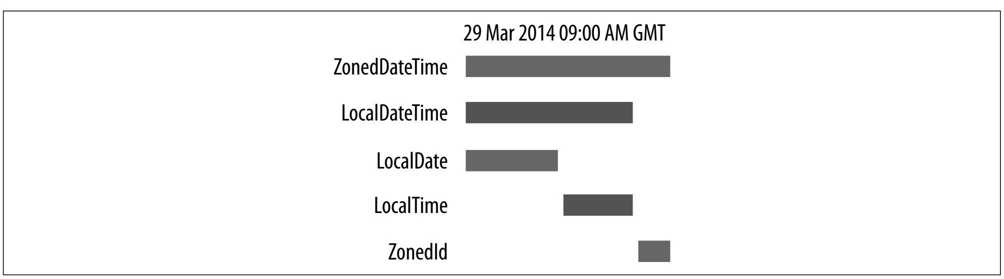


图 9-1：分解时间戳

关键是要知道，不同的地方适合使用不同的抽象方式。例如，有些商业应用主要处理的是 `LocalDate` 对象，此时需要的时间粒度是一个工作日。而有些应用需要亚秒级甚至是毫秒级精度。开发者要了解所需的业务逻辑，在应用中使用合适的表示方式。

2. 示例

日期和时间 API 不是一朝一夕就能完全掌握的。下面举个例子，这个示例定义了一个日志类，记录生日。如果碰巧你很容易忘记生日，那么这样的类（尤其是 `getBirthdaysInNextMonth()` 这样的方法）可以给你提供很大的帮助：

```
public class BirthdayDiary {
    private Map<String, LocalDate> birthdays;

    public BirthdayDiary() {
        birthdays = new HashMap<>();
    }

    public LocalDate addBirthday(String name, int day, int month,
                                 int year) {
        LocalDate birthday = LocalDate.of(year, month, day);
        birthdays.put(name, birthday);

        return birthday;
    }

    public LocalDate getBirthdayFor(String name) {
        return birthdays.get(name);
    }

    public int getAgeInYear(String name, int year) {
        Period period = Period.between(birthdays.get(name),
            birthdays.get(name).withYear(year));

        return period.getYears();
    }

    public Set<String> getFriendsOfAgeIn(int age, int year) {
        return birthdays.keySet().stream()
                .filter(p -> getAgeInYear(p, year) == age)
                .collect(Collectors.toSet());
    }

    public int getDaysUntilBirthday(String name) {
        Period period = Period.between(LocalDate.now(),
            birthdays.get(name));
        return period.getDays();
    }

    public Set<String> getBirthdaysIn(Month month) {
        return birthdays.entrySet().stream()
                .filter(p -> p.getValue().getMonth() == month)
                .map(p -> p.getKey())
                .collect(Collectors.toSet());
    }

    public Set<String> getBirthdaysInNextMonth() {
        return getBirthdaysIn(LocalDate.now().getMonth());
    }
```

```java
        public int getTotalAgeInYears() {
            return birthdays.keySet().stream()
                    .mapToInt(p -> getAgeInYear(p,
                          LocalDate.now().getYear()))
                    .sum();
        }
    }
```

这个类展示了如何使用低层 API 实现有用的功能。这个类还用到了一些新技术，例如 Java 的流 API，而且演示了如何把 `LocalDate` 类视作不可变的类使用，以及如何把日期当成值处理。

9.3.2 查询

很多情况下，我们要回答一些关于某个时间对象的问题，例如：

- 这个日期在 3 月 1 日之前吗？
- 这个日期所在的年份是闰年吗？
- 今天距我下一次生日还有多少天？

为了回答这些问题，可以使用 `TemporalQuery` 接口，其定义如下所示：

```java
public interface TemporalQuery<R> {
    R queryFrom(TemporalAccessor temporal);
}
```

`queryFrom()` 方法的参数不能为 `null`，不过，如果结果表示不存在的值，可以使用 `null` 作为返回值。

> `Predicate` 接口实现的查询可以理解为只能回答“是”或“否”的问题。而 `TemporalQuery` 接口实现的查询更普适，除了能回答“是”或“否”之外，还能回答“有多少”和“哪一个”等问题。

下面看一个查询的具体示例，这个查询回答的问题是：“这个日期在一年中的哪个季度？”Java 8 不直接支持季度，因此要使用类似下面的代码：

```java
LocalDate today = LocalDate.now();
Month currentMonth = today.getMonth();
Month firstMonthofQuarter = currentMonth.firstMonthOfQuarter();
```

这样写没有把季度单独抽象出来，还需要编写专用的代码。下面我们稍微扩展一下 JDK，定义如下的枚举类型：

```
public enum Quarter {
    FIRST, SECOND, THIRD, FOURTH;
}
```

现在，可以这样编写查询：

```
public class QuarterOfYearQuery implements TemporalQuery<Quarter> {
    @Override
    public Quarter queryFrom(TemporalAccessor temporal) {
        LocalDate now = LocalDate.from(temporal);

        if(now.isBefore(now.with(Month.APRIL).withDayOfMonth(1))) {
            return Quarter.FIRST;
        } else if(now.isBefore(now.with(Month.JULY)
                               .withDayOfMonth(1))) {
            return Quarter.SECOND;
        } else if(now.isBefore(now.with(Month.NOVEMBER)
                               .withDayOfMonth(1))) {
            return Quarter.THIRD;
        } else {
            return Quarter.FOURTH;
        }
    }
}
```

TemporalQuery 对象可以直接使用，也可以间接使用。下面各举一个例子：

```
QuarterOfYearQuery q = new QuarterOfYearQuery();

// 直接使用
Quarter quarter = q.queryFrom(LocalDate.now());
System.out.println(quarter);

// 间接使用
quarter = LocalDate.now().query(q);
System.out.println(quarter);
```

多数情况下，最好间接使用，即把查询对象作为参数传给 query() 方法，因为这样写出的代码更易于阅读。

9.3.3 调节器

调节器的作用是修改日期和时间对象。假如我们想获取某个时间戳所在季度的第一天：

```
public class FirstDayOfQuarter implements TemporalAdjuster {
    @Override
    public Temporal adjustInto(Temporal temporal) {

        final int currentQuarter = YearMonth.from(temporal)
                .get(IsoFields.QUARTER_OF_YEAR);

        switch (currentQuarter) {
```

```
            case 1:
                return LocalDate.from(temporal)
                        .with(TemporalAdjusters.firstDayOfYear());
            case 2:
                return LocalDate.from(temporal)
                        .withMonth(Month.APRIL.getValue())
                        .with(TemporalAdjusters.firstDayOfMonth());
            case 3:
                return LocalDate.from(temporal)
                        .withMonth(Month.JULY.getValue())
                        .with(TemporalAdjusters.firstDayOfMonth());
            case 4:
                return LocalDate.from(temporal)
                        .withMonth(Month.OCTOBER.getValue())
                        .with(TemporalAdjusters.firstDayOfMonth());
            default:
                return null; // 肯定不会执行到这里
        }
    }
}
```

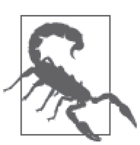
下面举个例子，看看如何使用调节器：

```
LocalDate now = LocalDate.now();
Temporal fdoq = now.with(new FirstDayOfQuarter());
System.out.println(fdoq);
```

这里的关键是 `with()` 方法，这段代码先读取一个 `Temporal` 对象，然后返回修改后的另一个对象。在处理不可变对象的 API 中经常会见到这种方式。

9.3.4　过时的日期和时间API

可惜，很多应用还没有转用 Java 8 中优秀的日期和时间库。所以，为了完整性，本节简要介绍一下以前的 Java 版本对日期和时间的支持（以 `java.util.Date` 类为基础）。



在 Java 8 环境中，别在使用过时的日期和时间类，尤其是 `java.util.Date` 类。

在较旧的 Java 版本中没有 `java.time` 包，开发者只能依赖 `java.util.Date` 类提供的基础支持。以前，这是表示时间戳的唯一方式。虽然这个类的名称是 `Date`，但其实它为日期和时间都提供了相应的组件，因此也为很多程序员带来了大量困扰。

`Date` 类提供的过时支持有很多问题。

- `Date` 类的实现方式不正确。它表示的其实不是日期，更像是时间戳。因此需要使用不同的方式表示日期、日期和时间，以及瞬时时间戳。

- `Date` 对象是可变的。创建日期的引用后，再次指向这个对象时可以修改它的值。
- `Date` 类不符合 ISO-8601 标准。这是全球通用的日期标准，规定什么是有效的日期。
- `Date` 类中有相当多的弃用方法。

这个版本的 JDK 使用两个构造方法创建 `Date` 对象：一个构造方法不接受参数，用于创建当前时间；另一个构造方法接受一个参数，即距 Epoch 时间 [5] 的毫秒数。

9.4 小结

本节介绍了多种不同类型的数据。文本和数字数据是最常见的，不过现实中的程序员还会遇到很多其他数据类型。下一章介绍存储数据的文件，以及处理 I/O 和网络的新方式。幸运的是，Java 为很多这种抽象都提供了良好的处理方式。

注 5：Epoch 时间是 1970-01-01 00:00:00 UTC。——译者注

第 10 章

处理文件和I/O

从第 1 版开始，Java 就支持输入 / 输出（I/O）。可是，由于 Java 极力想实现平台独立性，所以早期版本中的 I/O 功能更加强调可移植性而不是功能，因此不是那么好用。

本章后面会介绍，原来的 API 已经得到补充，变得很丰富，而且功能完善，易于使用。首先，我们要介绍 Java 以前处理 I/O 的“经典”方式，这是现代方式的基础。

10.1 Java处理I/O的经典方式

`File` 类是以前 Java 处理文件 I/O 的基础。这个抽象既能表示文件，也能表示目录，不过有时使用起来有些麻烦，写出的代码如下所示：

```
// 创建一个文件对象,表示用户的家目录
File homedir = new File(System.getProperty("user.home"));

// 创建一个对象,表示配置文件
//（家目录中应该存在这个文件）
File f = new File(homedir, "app.conf");

// 检查文件是否存在,是否真是文件,以及是否可读
if (f.exists() && f.isFile() && f.canRead()) {

  // 创建一个文件对象,表示新配置目录
  File configdir = new File(f, ".configdir");
  // 然后创建这个目录
  configdir.mkdir();
  // 最后,把配置文件移到新位置
  f.renameTo(new File(configdir, ".config"));
}
```

上述代码展现了 File 类使用灵活的一面，但也演示了这种抽象带来的一些问题。一般情况下，需要调用很多方法查询 File 对象才能判断这个对象到底表示的是什么，以及具有什么能力。

10.1.1 文件

File 类中有相当多的方法，但根本没有直接提供一些基本功能（尤其是无法读取文件的内容）。

下述代码简要总结了 File 类中的方法：

```
// 权限管理
boolean canX = f.canExecute();
boolean canR = f.canRead();
boolean canW = f.canWrite();

boolean ok;
ok = f.setReadOnly();
ok = f.setExecutable(true);
ok = f.setReadable(true);
ok = f.setWritable(false);

// 使用不同的方式表示文件名
File absF = f.getAbsoluteFile();
File canF = f.getCanonicalFile();
String absName = f.getAbsolutePath();
String canName = f.getCanonicalPath();
String name = f.getName();
String pName = getParent();
URI fileURI = f.toURI(); // 创建文件路径的URI形式

// 文件的元数据
boolean exists = f.exists();
boolean isAbs = f.isAbsolute();
boolean isDir = f.isDirectory();
boolean isFile = f.isFile();
boolean isHidden = f.isHidden();
long modTime = f.lastModified(); // 距Epoch时间的毫秒数
boolean updateOK = f.setLastModified(updateTime); // 毫秒
long fileLen = f.length();

// 文件管理操作
boolean renamed = f.renameTo(destFile);
boolean deleted = f.delete();

// 创建文件不会覆盖现有文件
boolean createdOK = f.createNewFile();

// 处理临时文件
File tmp = File.createTempFile("my-tmp", ".tmp");
tmp.deleteOnExit();
```

```
// 处理目录
boolean createdDir = dir.mkdir();
String[] fileNames = dir.list();
File[] files = dir.listFiles();
```

File 类中还有一些方法不完全符合这种抽象。其中多数方法都要查询文件系统（例如，查询可用空间）：

```
long free, total, usable;

free = f.getFreeSpace();
total = f.getTotalSpace();
usable = f.getUsableSpace();

File[] roots = File.listRoots(); // 所有可用的文件系统根目录
```

10.1.2 流

I/O 流抽象（不要跟 Java 8 集合 API 使用的流搞混了）出现在 Java 1.0 中，用于处理硬盘或其他源发出的连续字节流。

这个 API 的核心是一对抽象类，InputStream 和 OutputStream。这两个类使用广泛，事实上，“标准”输入和输出流（System.in 和 System.out）就是这种流。标准输入和输出流是 System 类的公开静态字段，在最简单的程序中也能用到：

```
System.out.println("Hello World!");
```

流的某些特定的子类，例如 FileInputStream 和 FileOutputStream，可以操作文件中单独的字节。例如，下述代码用于统计文件中 ASCII 97（小写的 a）出现的次数：

```
try (InputStream is = new FileInputStream("/Users/ben/cluster.txt")) {
  byte[] buf = new byte[4096];
  int len, count = 0;
  while ((len = is.read(buf)) > 0) {
    for (int i=0; i<len; i++)
      if (buf[i] == 97) count++;
  }
  System.out.println("'a's seen: "+ count);
} catch (IOException e) {
  e.printStackTrace();
}
```

使用这种方式处理硬盘中的数据缺乏灵活性，因为多数开发者习惯以字符而不是字节的方式思考问题。因此，这种流经常和高层的 Reader 和 Writer 类结合在一起使用。Reader 和 Writer 类处理的是字符流，而不是 InputStream 和 OutputStream 及其子类提供的低层字节流。

10.1.3 Reader和Writer类

把抽象从字节提升到字符后，开发者就更熟悉所面对的 API 了，而且这样也能规避很多由字符编码和 Unicode 等引起的问题。

`Reader` 和 `Writer` 类架构在字节流相关的类之上，无需再处理低层 I/O 流。这两个类有几个子类，往往都两两结合在一起使用，例如：

- `FileReader`
- `BufferedReader`
- `InputStreamReader`
- `FileWriter`
- `PrintWriter`
- `BufferedWriter`

若想读取一个文件中的所有行，并把这些行打印出来，可以在 `FileReader` 对象的基础上使用 `BufferedReader` 对象，如下述代码所示：

```
try (BufferedReader in =
  new BufferedReader(new FileReader(filename))) {
  String line;

  while((line = in.readLine()) != null) {
    System.out.println(line);
  }
} catch (IOException e) {
  // 这处理FileNotFoundException等异常
}
```

如果想从终端读取行，而不是文件，一般会在 `System.in` 对象上使用 `InputStreamReader` 对象。我们来看个例子，在这个示例中我们想从终端读取行，但特殊对待以特殊字符开头的行——这种行是要处理的命令（“元”），而不是普通文本。很多聊天程序，包括 IRC，都需要这种功能。这里，我们要借助第 9 章介绍的正则表达式：

```
Pattern SHELL_META_START = Pattern.compile("^#(\\w+)\\s*(\\w+)?");

try (BufferedReader console =
  new BufferedReader(new InputStreamReader(System.in))) {
  String line;

  READ: while((line = console.readLine()) != null) {
    // 检查特殊的命令
    Matcher m = SHELL_META_START.matcher(line);
    if (m.find()) {
      String metaName = m.group(1);
      String arg = m.group(2);
      doMeta(metaName, arg);
```

```
            continue READ;
          }

          System.out.println(line);
        }
      } catch (IOException e) {
        // 这里处理FileNotFoundException等异常
      }
```

若想把文本输出到文件中，可以使用如下代码：

```
File f = new File(System.getProperty("user.home")
 + File.separator + ".bashrc");
try (PrintWriter out
   = new PrintWriter(new BufferedWriter(new FileWriter(f)))) {
  out.println("## Automatically generated config file. DO NOT EDIT");
} catch (IOException iox) {
  // 处理异常
}
```

Java 处理 I/O 的旧风格中有些功能偶尔也有用。例如，处理文本文件时，`FilterInputStream` 类往往非常有用。对于想使用类似于经典"管道"I/O 方式通信的线程来说，Java 提供了 `PipedInputStream` 和 `PipedReader` 类，以及对应的写入器。

到目前为止，本章多次用到了一种语言特性——"处理资源的 `try` 语句"（try-with-resources，TWR）。这种语句的句法在 2.5.18 节简单介绍过，但要结合 I/O 等操作才能充分发挥潜能，而且还给旧 I/O 风格带来了新生。

10.1.4 再次介绍TWR

为了充分发挥 Java 的 I/O 能力，一定要理解如何以及何时使用 TWR。何时使用很好确定，只要可以用就用。

在 TWR 出现之前，必须手动关闭资源，而且处理资源之间复杂交互的代码可能有缺陷，无法关闭资源，从而导致资源泄露。

事实上，根据甲骨文工程师的估计，在 JDK 6 的初始版本中，处理资源的代码有 60% 都不正确。因此，既然连平台的作者都无法完全正确地手动处理资源，那么所有新代码显然都应该使用 TWR。

实现 TWR 的关键是一个新接口——`AutoCloseable`。这个新接口（在 Java 7 中出现）是 `Closeable` 的直接超接口，表示资源必须自动关闭。为此，编译器会插入特殊的异常处理代码。

在 TWR 的资源子句中，只能声明实现了 `AutoCloseable` 接口的对象，而且数量不限：

```
try (BufferedReader in = new BufferedReader(
                           new FileReader("profile"));
     PrintWriter out = new PrintWriter(
                          new BufferedWriter(
                           new FileWriter("profile.bak")))) {
  String line;
  while((line = in.readLine()) != null) {
    out.println(line);
  }
} catch (IOException e) {
  // 这里处理FileNotFoundException等异常
}
```

这样写，资源的作用域就自动放入 try 块中，各个资源（不管是可读的还是可写的）会按照正确的顺序自动关闭，而且编译器插入的异常处理代码会考虑到资源之间的相互依赖关系。

TWR 的作用大致和 C# 的 using 关键字类似，开发者可以把 TWR 看成“正确的终结方式”。6.4 节说过，新代码绝对不能直接使用终结机制，而一定要使用 TWR。旧代码应该根据情况尽早重构，换用 TWR。

10.1.5　I/O经典处理方式的问题

即便添加了受欢迎的 TWR，File 及相关的类还是有一些问题，就算执行标准的 I/O 操作也不理想，无法广泛使用。例如：

- 缺少处理常见操作的方法；
- 在不同的平台中不能使用一致的方式处理文件名；
- 没有统一的文件属性模型（例如，读写模型）；
- 难以遍历未知的目录结构；
- 没有平台或操作系统专用的特性；
- 不支持使用非阻塞方式处理文件系统。

为了改善这些缺点，Java 的 I/O API 在过去的几个主版本中一直在改进。直到 Java 7，处理 I/O 才真正变得简单而高效。

10.2　Java处理I/O的现代方式

Java 7 引入了全新的 I/O API（一般称为 NIO.2），几乎可以完全取代以前使用 File 类处理 I/O 的方式。新添加的各个类都在 java.nio.file 包中。

很多情况下，使用 Java 7 引入的新 API 处理 I/O 更简单。新 API 分为两大部分：第一部分是一个新抽象，Path 接口（这个接口的作用可以理解为表示文件的位置，这个位置可以有内容，也可以没有）；第二部分是很多处理文件和文件系统的新方法，方便且实用。这些

新方法都是 Files 类的静态方法。

10.2.1 文件

例如，使用 Files 类的新功能执行基本的复制操作非常简单，如下所示：

```
File inputFile = new File("input.txt");
try (InputStream in = new FileInputStream(inputFile)) {
  Files.copy(in, Paths.get("output.txt"));
} catch(IOException ex) {
  ex.printStackTrace();
}
```

下面我们纵览一下 Files 类中的一些重要方法，多数方法执行的操作都不言自明。很多情况下，这些方法都有返回类型。不过，除了人为的个例，或者重复等效 C 代码的行为，很少使用返回类型。

```
Path source, target;
Attributes attr;
Charset cs = StandardCharsets.UTF_8;

// 创建文件
//
// 示例路径 --> /home/ben/.profile
// 示例属性 --> rw-rw-rw-
Files.createFile(target, attr);

// 删除文件
Files.delete(target);
boolean deleted = Files.deleteIfExists(target);

// 复制/移动文件
Files.copy(source, target);
Files.move(source, target);

// 读取信息的实用方法
long size = Files.size(target);

FileTime fTime = Files.getLastModifiedTime(target);
System.out.println(fTime.to(TimeUnit.SECONDS));

Map<String, ?> attrs = Files.readAttributes(target, "*");
System.out.println(attrs);

// 处理文件类型的方法
boolean isDir = Files.isDirectory(target);
boolean isSym = Files.isSymbolicLink(target);

// 处理读写操作的方法
List<String> lines = Files.readAllLines(target, cs);
byte[] b = Files.readAllBytes(target);
```

```
BufferedReader br = Files.newBufferedReader(target, cs);
BufferedWriter bwr = Files.newBufferedWriter(target, cs);

InputStream is = Files.newInputStream(target);
OutputStream os = Files.newOutputStream(target);
```

Files 类中的某些方法可以接受可选的参数，为方法执行的操作指定其他行为（可能是针对特定实现的行为）。

这个 API 的某些决策偶尔会导致让人烦恼的行为。例如，默认情况下，复制操作不会覆盖已经存在的文件，所以需要使用一个复制选项指定这种行为：

```
Files.copy(Paths.get("input.txt"), Paths.get("output.txt"),
           StandardCopyOption.REPLACE_EXISTING);
```

StandardCopyOption 是一个枚举，实现了 CopyOption 接口。而且，LinkOption 枚举也实现了 CopyOption 接口。所以，Files.copy() 方法能接受任意个 LinkOption 或 StandardCopyOption 参数。LinkOption 用于指定如何处理符号链接（当然，前提是底层操作系统支持符号链接）。

10.2.2 路径

Path 接口可用于在文件系统中定位文件。这个接口表示的路径具有下述特性：

- 系统相关
- 有层次结构
- 由一系列路径元素组成
- 假设的（可能还不存在，或者已经删除）

因此，Path 对象和 File 对象完全不同。其中特别重要的一点是，Path 是接口，而不是类，这体现了系统相关性。因此，不同的文件系统提供方可以使用不同的方式实现 Path 接口，提供系统专用的特性，但同时还保有整体的抽象。

组成 Path 对象的元素中有一个可选的根组件，表示实例所属文件系统的层次结构。注意，有些 Path 对象可能没有根组件，例如表示相对路径的 Path 对象。除了根组件之外，每个 Path 实例都有零个或多个目录名和名称元素。

名称元素是离目录层次结构的根最远的元素，表示文件或目录的名称。Path 对象的内容可以理解为使用特殊的分隔符把各个路径元素联接在一起。

Path 对象是个抽象概念，和任何物理文件路径都没关联。因此，可以轻易表示还不存在的文件路径。Java 提供的 Paths 类中有创建 Path 实例的工厂方法。

Paths 类提供了两个 get() 方法，用于创建 Path 对象。普通的版本接受一个 String 对象，

使用默认的文件系统提供方。另一个版本接受一个 URI 对象，利用了 NIO.2 能插入其他提供方定制文件系统的特性。这是高级用法，有兴趣的开发者可以参阅相关文档。

```
Path p = Paths.get("/Users/ben/cluster.txt");
Path p = Paths.get(new URI("file:///Users/ben/cluster.txt"));
System.out.println(p2.equals(p));

File f = p.toFile();
System.out.println(f.isDirectory());
Path p3 = f.toPath();
System.out.println(p3.equals(p));
```

这个示例还展示了 Path 对象和 File 对象之间可以轻易地相互转换。有了 Path 类中的 toFile() 方法和 File 类中的 toPath() 方法，开发者可以毫不费力地在两个 API 之间切换，而且可以使用一种直观的方式重构使用 File 类的代码，换用 Path 接口。

除此之外，还可以使用 Files 类中一些有用的“桥接”方法。通过这些方法可以轻易使用旧的 I/O API，例如，有些便利方法可以创建 Writer 对象，把内容写入 Path 对象指定的位置：

```
Path logFile = Paths.get("/tmp/app.log");
try (BufferedWriter writer =
       Files.newBufferedWriter(logFile, StandardCharsets.UTF_8,
                               StandardOpenOption.WRITE)) {
  writer.write("Hello World!");
  // ...
} catch (IOException e) {
  // ...
}
```

这里使用了 StandardOpenOption 枚举，其作用和复制选项类似，不过用于指定打开新文件的行为。

在这个示例中，我们使用 Path API 完成了下述操作：

- 创建一个 Path 对象，对应于一个新文件；
- 使用 Files 类创建那个新文件；
- 创建一个 Writer 对象，打开那个文件；
- 把内容写入那个文件；
- 写入完毕后自动关闭那个文件。

下面再举个例子。这个示例基于前面的代码，把一个 .jar 文件本身当成 FileSystem 对象处理，直接把一个新文件添加到这个 JAR 文件中。JAR 文件其实就是 ZIP 文件，所以这种技术也适用于 .zip 压缩文件。

```
Path tempJar = Paths.get("sample.jar");
try (FileSystem workingFS =
```

```
    FileSystems.newFileSystem(tempJar, null)) {
    Path pathForFile = workingFS.getPath("/hello.txt");
    List<String> ls = new ArrayList<>();
    ls.add("Hello World!");

    Files.write(pathForFile, ls, Charset.defaultCharset(),
                StandardOpenOption.WRITE, StandardOpenOption.CREATE);
}
```

这个示例展示了如何使用 `getPath()` 方法在 `FileSystem` 对象中创建 `Path` 对象。使用这种技术，开发者其实可以把 `FileSystem` 对象当成黑盒。

Java 最初提供的 I/O API 受到的批评之一是，不支持本地 I/O 和高性能 I/O。Java 1.4 首次对此提出了解决方案——New I/O (NIO) API，而且在后续版本中一直在改善。

10.3 NIO中的通道和缓冲区

NIO 中的缓冲区是对高性能 I/O 的一种低层抽象，为指定基本类型组成的线性序列提供容器。后面的示例都以处理 `ByteBuffer` 对象（最常见）为例。

10.3.1 ByteBuffer对象

`ByteBuffer` 对象是字节序列，理论上，在注重性能的场合中可以代替 `byte[]` 类型的数组。为了得到最好的性能，`ByteBuffer` 支持直接使用 JVM 所在平台提供的本地功能处理缓冲区。

这种方式叫作“直接缓冲区”，只要可能就会绕过 Java 堆内存。直接缓冲区在本地内存中分配，而不是在标准的 Java 堆内存中。而且，垃圾回收程序对待直接缓冲区的方式和普通的堆中 Java 对象不同。

若想创建 `ByteBuffer` 类型的直接缓冲区对象，可以调用工厂方法 `allocateDirect()`。除此之外，还有 `allocate()` 方法，用于创建堆中缓冲区，不过现实中不常使用。

创建字节缓冲区的第三种方式是打包现有的 `byte[]` 数组。这种方式创建的是堆中缓冲区，目的是以更符合面向对象的方式处理底层字节：

```
ByteBuffer b = ByteBuffer.allocateDirect(65536);
ByteBuffer b2 = ByteBuffer.allocate(4096);

byte[] data = {1, 2, 3};
ByteBuffer b3 = ByteBuffer.wrap(data);
```

字节缓冲区只能使用低层方式访问字节，因此开发者要手动处理细节，例如需要处理字节的字节顺序和 Java 整数基本类型的符号：

```
b.order(ByteOrder.BIG_ENDIAN);

int capacity = b.capacity();
int position = b.position();
int limit = b.limit();
int remaining = b.remaining();
boolean more = b.hasRemaining();
```

把数据存入缓冲区或从缓冲区中取出有两种操作方式：一种是单值操作，一次读写一个值；另一种是批量操作，一次读写一个 byte[] 数组或 ByteBuffer 对象，处理多个值（可能很多）。使用批量操作才能获得预期的性能提升：

```
b.put((byte)42);
b.putChar('x');
b.putInt(0xcafebabe);

b.put(data);
b.put(b2);

double d = b.getDouble();
b.get(data, 0, data.length);
```

单值形式还支持直接处理缓冲区中绝对位置上的数据：

```
b.put(0, (byte)9);
```

缓冲区这种抽象只存在于内存中，如果想影响外部世界（例如文件或网络），需要使用 Channel（通道）对象。Channel 接口在 java.nio.channels 包中定义，表示支持读写操作的实体连接。文件和套接字是两种常见的通道，不过我们要意识到，用于低延迟数据处理的自定义实现也属于通道。

通道在创建时处于打开状态，随后可以将其关闭。一旦关闭，就无法再打开。一般来说，通道要么可读要么可写，不能既可读又可写。若想理解通道，关键是要知道：

- 从通道中读取数据时会把字节存入缓冲区
- 把数据写入通道时会从缓冲区中读取字节

假如我们要计算一个大文件前 16M 数据片段的校验和：

```
FileInputStream fis = getSomeStream();
boolean fileOK = true;

try (FileChannel fchan = fis.getChannel()) {
  ByteBuffer buffy = ByteBuffer.allocateDirect(16 * 1024 * 1024);
  while(fchan.read(buffy) != -1 || buffy.position() > 0 || fileOK) {
    fileOK = computeChecksum(buffy);
    buffy.compact();
  }
} catch (IOException e) {
```

```
    System.out.println("Exception in I/O");
  }
```

上述代码会尽量使用本地 I/O，不会把字节大量复制进出 Java 堆内存。如果 computeChecksum() 方法实现得好，上述代码的性能就很高。

10.3.2 映射字节缓冲区

这是一种直接字节缓冲区，包含一个内存映射文件（或内存映射文件的一部分）。这种缓冲区由 FileChannel 对象创建，不过要注意，内存映射操作之后决不能使用 MappedByteBuffer 对象对应的 File 对象，否则会抛出异常。为了规避这种问题，我们可以使用处理资源的 try 语句，严格限制相关对象的作用域：

```
try (RandomAccessFile raf =
  new RandomAccessFile(new File("input.txt"), "rw");
     FileChannel fc = raf.getChannel();) {

  MappedByteBuffer mbf =
    fc.map(FileChannel.MapMode.READ_WRITE, 0, fc.size());
  byte[] b = new byte[(int)fc.size()];
  mbf.get(b, 0, b.length);
  for (int i=0; i<fc.size(); i++) {
    b[i] = 0; // 这是一个副本,不会写入原文件
  }
  mbf.position(0);
  mbf.put(b); // 清空文件
}
```

就算有了缓冲区，Java 在单个线程中同步执行大型 I/O 操作（例如，在文件系统之间传输 10G 数据）时还是会遇到一些限制。在 Java 7 之前，遇到这种操作时往往要自己编写多线程代码，而且要管理一个单独的线程执行后台复制操作。下面介绍 JDK 7 新添加的异步 I/O 功能。

10.4 异步I/O

新异步功能的关键组成部分是一些实现 Channel 接口的类，这些类可以处理需要交给后台线程完成的 I/O 操作。这种功能还可以应用于长期运行的大型操作和其他几种场合。

这一节专门介绍处理文件 I/O 的 AsynchronousFileChannel 类，除此之外还要了解一些其他异步通道。本章末尾会介绍异步套接字。这一节介绍的内容包括：

- 使用 AsynchronousFileChannel 类处理文件 I/O
- 使用 AsynchronousSocketChannel 类处理客户端套接字 I/O
- 使用 AsynchronousServerSocketChannel 类处理能接受连入连接的异步套接字

和异步通道交互有两种不同的方式：使用 Future 接口的方式和回调方式。

10.4.1 基于Future接口的方式

第 11 章会详细介绍 Future 接口，现在你只需知道这个接口表示进行中的任务，可能已经完成，也可能还未完成。这个接口有两个关键的方法。

- isDone()

  返回布尔值，表示任务是否已经完成。

- get()

  返回结果。如果已经结束，立即返回；如果还未结束，在完成前一直阻塞。

下面看个示例程序。这个程序异步读取一个大型文件（可能有 100 Mb）：

```
try (AsynchronousFileChannel channel =
         AsynchronousFileChannel.open(Paths.get("input.txt"))) {
  ByteBuffer buffer = ByteBuffer.allocateDirect(1024 * 1024 * 100);
  Future<Integer> result = channel.read(buffer, 0);

  while(!result.isDone()) {
    // 做些其他有用的操作……
  }

  System.out.println("Bytes read: " + result.get());
}
```

10.4.2 基于回调的方式

处理异步 I/O 的回调方式基于 CompletionHandler 接口实现，这个接口定义了两个方法，completed() 和 failed()，分别在操作成功和失败时调用。

处理异步 I/O 时，如果想立即收到事件提醒，可以使用这种方式。例如，有大量 I/O 操作要执行，但其中某次操作失败不会导致重大错误，这种情况就可以使用回调方式：

```
byte[] data = {2, 3, 5, 7, 11, 13, 17, 19, 23};
ByteBuffer buffy = ByteBuffer.wrap(data);

CompletionHandler<Integer,Object> h =
  new CompletionHandler() {
  public void completed(Integer written, Object o) {
    System.out.println("Bytes written: " + written);
  }

  public void failed(Throwable x, Object o) {
    System.out.println("Asynch write failed: "+ x.getMessage());
  }
};

try (AsynchronousFileChannel channel =
       AsynchronousFileChannel.open(Paths.get("primes.txt"),
```

```
            StandardOpenOption.CREATE, StandardOpenOption.WRITE)) {

    channel.write(buffy, 0, null, h);
    Thread.sleep(1000); // 必须这么做,防止退出太快
  }
```

AsynchronousFileChannel 对象关联一个后台线程池，所以处理 I/O 操作时，原线程可以继续处理其他任务。

默认情况下，这个线程池由运行时提供并管理。如果需要，线程池也可以由应用创建和管理（通过 AsynchronousFileChannel.open() 方法的某个重载形式），不过一般不需要这么做。

最后，为了完整性，我们还要简单介绍 NIO 对多路复用 I/O 的支持。在多路复用 I/O 中，单个线程能管理多个通道，而且会检测哪个通道做好了读或写的准备。支持多路复用 I/O 的类在 java.nio.channels 包中，包括 SelectableChannel 和 Selector。

编写需要高伸缩性的高级应用时，这种非阻塞式多路复用技术特别有用，不过对这个话题的完整讨论超出了本书的范畴。

10.4.3 监视服务和目录搜索

我们要介绍的最后一种异步服务是监视目录，或访问目录（或树状结构）。监视服务会观察目录中发生的所有事情，例如创建或修改文件：

```
try {
  WatchService watcher = FileSystems.getDefault().newWatchService();

  Path dir = FileSystems.getDefault().getPath("/home/ben");
  WatchKey key = dir.register(watcher,
                      StandardWatchEventKinds.ENTRY_CREATE,
                      StandardWatchEventKinds.ENTRY_MODIFY,
                      StandardWatchEventKinds.ENTRY_DELETE);

  while(!shutdown) {
    key = watcher.take();
    for (WatchEvent<?> event: key.pollEvents()) {
      Object o = event.context();
      if (o instanceof Path) {
        System.out.println("Path altered: "+ o);
      }
    }
    key.reset();
  }
}
```

相比之下，目录流提供的是单个目录中当前所有文件的情况。例如，若想列出所有 Java 源码文件及其大小（以字节为单位），可以使用如下代码：

```
try(DirectoryStream<Path> stream =
    Files.newDirectoryStream(Paths.get("/opt/projects"), "*.java")) {
  for (Path p : stream) {
    System.out.println(p +": "+ Files.size(p));
  }
}
```

这个 API 有个缺点，即只能返回匹配通配模式的元素，这有时不够灵活。我们可以更进一步，使用新方法 `Files.find()` 和 `Files.walk()`，递归遍历目录，找出每个元素：

```
final Pattern isJava = Pattern.compile(".*\\.java$");
final Path homeDir = Paths.get("/Users/ben/projects/");
Files.find(homeDir, 255,
  (p, attrs) -> isJava.matcher(p.toString()).find())
     .forEach(q -> {System.out.println(q.normalize());});
```

我们还可以更进一步，使用 `java.nio.file` 包中的 `FileVisitor` 接口编写高级的解决方案，不过，此时需要开发者实现 `FileVisitor` 接口中的全部四个方法，不能像上述代码那样只使用一个 lambda 表达式。

本章的最后一节要讨论 Java 对网络的支持以及 JDK 中相应的核心类。

10.5 网络

Java 平台支持大量标准的网络协议，因此编写简单的网络应用非常容易。Java 对网络支持的核心 API 在 `java.net` 包中，其他扩展 API 则由 `javax.net` 包（尤其是 `javax.net.ssl` 包）提供。

开发应用时最易于使用的协议是超文本传输协议（HyperText Transmission Protocol，HTTP），这个协议是 Web 的基础通信协议。

10.5.1 HTTP

HTTP 是 Java 原生支持的最高层网络协议。这个协议非常简单，基于文本，在 TCP/IP 标准协议族的基础上实现。HTTP 可以在任何网络端口中使用，不过通常使用 80 端口。

`URL` 是关键的类——这个类原生支持 `http://`、`ftp://`、`file://` 和 `https://` 形式的 URL。这个类使用起来非常简单，最简单的示例是下载指定 URL 对应页面的内容。在 Java 8 中，使用下面的代码即可：

```
URL url = new URL("http://www.jclarity.com/");
try (InputStream in = url.openStream()) {
  Files.copy(in, Paths.get("output.txt"));
} catch(IOException ex) {
  ex.printStackTrace();
}
```

若想深入低层控制，例如获取请求和响应的元数据，可以使用 URLConnection 类，编写如下代码：

```
try {
  URLConnection conn = url.openConnection();

  String type = conn.getContentType();
  String encoding = conn.getContentEncoding();
  Date lastModified = new Date(conn.getLastModified());
  int len = conn.getContentLength();
  InputStream in = conn.getInputStream();
} catch (IOException e) {
  // 处理异常
}
```

HTTP 定义了多个“请求方法”，客户端使用这些方法操作远程资源。这些方法是：

GET、POST、HEAD、PUT、DELETE、OPTIONS、TRACE

各个方法的用法稍微不同，例如：

- GET 只能用于取回文档，不能执行任何副作用；
- HEAD 和 GET 作用一样，但是不返回主体——如果程序只想检查 URL 对应的网页是否有变化，可以使用 HEAD；
- 如果想把数据发送给服务器处理，要使用 POST。

默认情况下，Java 始终使用 GET 方法，不过也提供了使用其他方法的方式，用于开发更复杂的应用。然而，这需要做一些额外工作。在下面这个示例中，我们使用 BBC 网站提供的搜索功能搜索关于 Java 的新闻：

```
URL url = new URL("http://www.bbc.co.uk/search");

String rawData = "q=java";
String encodedData = URLEncoder.encode(rawData, "ASCII");
String contentType = "application/x-www-form-urlencoded";

HttpURLConnection conn = (HttpURLConnection) url.openConnection();
conn.setInstanceFollowRedirects(false);
conn.setRequestMethod("POST");
conn.setRequestProperty("Content-Type", contentType );
conn.setRequestProperty("Content-Length",
  String.valueOf(encodedData.length()));

conn.setDoOutput(true);
OutputStream os = conn.getOutputStream();
os.write( encodedData.getBytes() );

int response = conn.getResponseCode();
if (response == HttpURLConnection.HTTP_MOVED_PERM
    || response == HttpURLConnection.HTTP_MOVED_TEMP) {
```

```
      System.out.println("Moved to: "+ conn.getHeaderField("Location"));
    } else {
      try (InputStream in = conn.getInputStream()) {
        Files.copy(in, Paths.get("bbc.txt"),
                    StandardCopyOption.REPLACE_EXISTING);
      }
    }
```

注意，请求参数要在请求的主体中发送，而且发送前要编码。我们还要禁止跟踪 HTTP 重定向，手动处理服务器返回的每个重定向响应。这是因为 `HttpURLConnection` 类有个缺陷，不能正确处理 POST 请求的重定向响应。

开发这种高级 HTTP 应用时，多数情况下开发者一般都会使用专门的 HTTP 客户端库，例如 Apache 提供的那个库，而不会使用 JDK 提供的类从零编写所有代码。

下面介绍网络协议栈的下一层，传输控制协议（Transmission Control Protocol，TCP）。

10.5.2 TCP

TCP 是互联网中可靠传输网络数据的基础，确保传输的网页和其他互联网流量完整且易于理解。从网络理论的视角来看，由于 TCP 具有下述特性，才能作为互联网流量的“可靠性层”。

- 基于连接

  数据属于单个逻辑流（连接）。

- 保证送达

  如果未收到数据包，会一直重新发送，直到送达为止。

- 错误检查

  能检测到网络传输导致的损坏，并自动修复。

TCP 是双向通信通道，使用特殊的编号机制（TCP 序号）为数据块指定序号，确保通信流的两端保持同步。为了在同一个网络主机中支持多个不同的服务，TCP 使用端口号识别服务，而且能确保某个端口的流量不会走另一个端口传输。

Java 使用 `Socket` 和 `ServerSocket` 类表示 TCP。这两个类分别表示连接中的客户端和服务器端。也就是说，Java 既能连接网络服务，也能用来实现新服务。

举个例子，我们来重新实现 HTTP。这个协议基于文本，相对简单。连接的两端都要实现，下面先基于 TCP 套接字实现 HTTP 客户端。为此，其实我们需要实现 HTTP 协议的细节，不过我们有个优势——完全掌控着 TCP 套接字。

我们既要从客户端套接字中读取数据，也要把数据写入客户端套接字，而且构建请求时要

遵守 HTTP 标准（RFC 2616）。最终写出的代码如下所示：

```
String hostname = "www.example.com";
int port = 80;
String filename = "/index.html";

try (Socket sock = new Socket(hostname, port);
  BufferedReader from = new BufferedReader(
      new InputStreamReader(sock.getInputStream()));
  PrintWriter to = new PrintWriter(
      new OutputStreamWriter(sock.getOutputStream())); ) {

  // HTTP协议
  to.print("GET " + filename +
    " HTTP/1.1\r\nHost: "+ hostname +"\r\n\r\n");
  to.flush();

  for(String l = null; (l = from.readLine()) != null; )
    System.out.println(l);

}
```

在服务器端，可能需要处理多个连入连接。因此，需要编写一个服务器主循环，然后使用 accept() 方法从操作系统中接收一个新连接。随后，要迅速把这个新连接传给单独的类处理，好让服务器主循环继续监听新连接。服务器端的代码比客户端复杂：

```
// 处理连接的类
private static class HttpHandler implements Runnable {
  private final Socket sock;
  HttpHandler(Socket client) { this.sock = client; }

  public void run() {
    try (BufferedReader in =
           new BufferedReader(
             new InputStreamReader(sock.getInputStream()));
         PrintWriter out =
           new PrintWriter(
             new OutputStreamWriter(sock.getOutputStream())); ) {
      out.print("HTTP/1.0 200\r\nContent-Type: text/plain\r\n\r\n");
      String line;
      while((line = in.readLine()) != null) {
        if (line.length() == 0) break;
        out.println(line);
      }
    } catch(Exception e) {
      // 处理异常
    }
  }
}

// 服务器主循环
public static void main(String[] args) {
  try {
```

```
    int port = Integer.parseInt(args[0]);

    ServerSocket ss = new ServerSocket(port);
    for(;;) {
      Socket client = ss.accept();
      HTTPHandler hndlr = new HTTPHandler(client);
      new Thread(hndlr).start();
    }
  } catch (Exception e) {
    // 处理异常
  }
}
```

为通过 TCP 通信的应用设计协议时，要谨记一个简单而意义深远的网络架构原则——Postel 法则（以互联网之父之一 Jon Postel 的名字命名）。这个法则有时表述为："发送时要保守，接收时要开放。"这个简单的原则表明，网络系统中的通信有太多可能性，即便非常不完善的实现也是如此。

如果开发者遵守 Postel 法则，还遵守尽量保持协议简单这个通用原则（有时也叫 KISS 原则[1]），那么，基于 TCP 的通信实现起来要比不遵守时更简单。

TCP 下面是互联网通用的运输协议——互联网协议（Internet Protocol，IP）。

10.5.3 IP

IP 是传输数据的最低层标准，抽象了把字节从 A 设备移动到 B 设备的物理网络技术。

和 TCP 不同，IP 数据包不能保证一定送达，在传输的路径中，任何过载的系统都可能会丢掉数据包。IP 数据包有目的地，但一般没有路由数据——真正传送数据的是沿线的物理传输介质（可能有多种不同的介质）。

在 Java 中可以创建基于单个 IP 数据包（首部除了可以指定使用 TCP 协议，还可以指定使用 UDP[2] 协议）的数据报服务，不过，除了延迟非常低的应用之外很少需要这么做。Java 使用 `DatagramSocket` 类实现这种功能，不过很少有开发者需要深入到网络协议栈的这一层。

最后，值得注意的是，互联网使用的寻址方案目前正在经历一些变化。目前使用的 IP 版本是 IPv4，可用的网络地址有 32 位存储空间。现在，这个空间严重不足，因此已经开始部署多种缓解技术。

IP 的下一版（IPv6）已经出现，但还没广泛使用。不过，在未来十年，IPv6 应该会更为普及。令人欣慰的是，Java 已经对这种新寻址方案提供了良好支持。

注 1：KISS 是"Keep it simple, stupid"的简称。——译者注

注 2：UDP 是 User Datagram Protocol 的简称。——译者注

第 11 章

类加载、反射和方法句柄

第 3 章提到过 Java 的 Class 对象，这是在运行中的 Java 进程里表示实时类型的方式。本章以此为基础，讨论 Java 环境加载并让新类型可用的方式。本章后半部分介绍 Java 的内省功能——既会介绍最初的反射 API，也会介绍较新的方法句柄功能。

11.1　类文件、类对象和元数据

第 1 章说过，类文件是编译 Java 源码文件（也可能是其他语言的源码文件）得到的中间格式，供 JVM 使用。类文件是二进制文件，目的不是供人类阅读。

运行时通过包含元数据的类对象表示类文件，而类对象表示的是从中创建类文件的 Java 类型。

11.1.1　类对象示例

在 Java 中，获取类对象有多种方式。其中最简单的方式是：

```
Class<?> myCl = getClass();
```

上述代码返回调用 getClass() 方法的实例对应的类对象。查看 Object 类的公开方法之后我们知道，Object 类中的 getClass() 方法是公开的，所以，可以获取任意对象 o 的类对象：

```
Class<?> c = o.getClass();
```

已知类型的类对象还可以写成“类字面量”：

```
// 类型名称后面加上".class",表示的是类字面量
c = int.class; // 等同于Integer.TYPE
c = String.class; // 等同于"a string".getClass()
c = byte[].class; // 字节数组的类型
```

基本类型和 void 也能使用字面量表示类对象：

```
// 使用预先定义好的常量获取基本类型的类对象
c = Void.TYPE; // 特殊的"没有返回值"类型
c = Byte.TYPE; // 表示byte类型的类对象
c = Integer.TYPE; // 表示int类型的类对象
c = Double.TYPE;  // Short、Character、Long、Float等类型也可以这么做
```

对于未知的类型，要使用更复杂的方法。

11.1.2　类对象和元数据

类对象包含指定类型的元数据，包括这个类中定义的方法、字段和构造方法等。开发者可以使用这些元数据审查类，就算加载类时对这个类一无所知也可以审查。

例如，可以找出类文件中所有的弃用方法（弃用方法使用 @Deprecated 注解标记）：

```
Class<?> clz = getClassFromDisk();
for (Method m : clz.getMethods()) {
  for (Annotation a : m.getAnnotations()) {
    if (a.annotationType() == Deprecated.class) {
      System.out.println(m.getName());
    }
  }
}
```

我们还可以找出两个类文件的共同祖先类。下面这种简单的写法在使用同一个类加载程序加载两个类时才能使用：

```
public static Class<?> commonAncestor(Class<?> cl1, Class<?> cl2) {
  if (cl1 == null || cl2 == null) return null;
  if (cl1.equals(cl2)) return cl1;
  if (cl1.isPrimitive() || cl2.isPrimitive()) return null;

  List<Class<?>> ancestors = new ArrayList<>();
  Class<?> c = cl1;
  while (!c.equals(Object.class)) {
    if (c.equals(cl2)) return c;
    ancestors.add(c);
    c = c.getSuperclass();
  }
  c = cl2;
  while (!c.equals(Object.class)) {
    for (Class<?> k : ancestors) {
      if (c.equals(k)) return c;
    }
```

```
        c = c.getSuperclass();
      }

      return Object.class;
    }
```

类文件必须符合非常明确的布局才算合法，JVM 才能加载。类文件包含以下部分（按如下顺序）：

- 魔法数（所有类文件都以 CA FE BA BE 这四个十六进制的字节开始）
- 使用的类文件标准版本
- 当前类的常量池
- 访问标志（abstract、public 等）
- 当前类的名称
- 继承信息（例如超类的名称）
- 实现的接口
- 字段
- 方法
- 属性

类文件是简单的二进制格式，不过人类不可读。如果想了解其中的内容，要使用 javap（参见第 13 章）等工具。

类文件中最常使用的部分之一是常量池。常量池中包含类需要引用的所有方法、类、字段和常量（不管在不在当前类中）。常量池经过精心设计，字节码通过索引序号就能方便地引用其中的条目——这么做节省了字节码占用的空间。

不同的 Java 版本生成的类文件版本有所不同，不过，Java 向后兼容的规则之一是，新版 JVM（及其他工具）都能使用旧版类文件。

下面介绍在类加载过程中如何使用硬盘中的字节新建类对象。

11.2　类加载的各个阶段

类加载是把新类型添加到运行中的 JVM 进程里的过程。这是新代码进入 Java 系统的唯一方式，也是 Java 平台中把数据变成代码的唯一方式。类加载分为几个阶段，下面一一介绍。

11.2.1　加载

类加载过程首先会加载一个字节数组。这个数组往往从文件系统中读取，不过也可以从

URL 或其他位置（一般使用 `Path` 对象表示）读取。

`Classloader::defineClass()` 方法的作用是把类文件（表示为字节数据）转换成类对象。这是受保护的方法，因此不通过子类无法访问。

`defineClass()` 的第一个任务是加载。加载的过程中会生成类对象的骨架，对应于尝试加载的类。这个阶段会对类做些基本检查（例如，会检查常量池中的常量，确保前后一致）。

不过，加载阶段不会生成完整的类对象，而且类也还不能使用。加载结束后，必须链接类。这一步细分为几个子阶段：

- 验证
- 准备和解析
- 初始化

11.2.2　验证

验证阶段确认类文件与预期相符，而且没有违背 JVM 的安全模型（详情参见 11.3 节）。

JVM 字节码经过精心设计，（几乎）可以静态检查。这么做会减慢类加载过程，不过能加快运行时（因为此时可以不做检查）。

验证阶段的目的是避免 JVM 执行可能导致自身崩溃的字节码，或者把 JVM 带入未测试的未知状态，出现恶意代码能攻击的漏洞。验证字节码能防御恶意编写的 Java 字节码，还能防止不信任的 Java 编译器输出无效的字节码。

> 默认方法机制在类加载过程中能正常运作。加载接口的实现时，会检查是否实现了默认方法，如果实现了，类加载过程正常向下运行；如果未实现，则为实现接口的类打补丁，添加缺失方法的默认实现。

11.2.3　准备和解析

验证通过后，类就做好了使用的准备。内存分配好了，类中的静态变量也准备初始化了。

在这个阶段，变量还未初始化，而且也没执行新类的字节码。开始运行代码之前，JVM 要确保运行时知道这个类文件引用的每个类型。如果不知道，可能还要加载这些类型——再开始其他类加载过程，让 JVM 加载新类型。

这个加载和发现的过程可能会不断进行下去，直到知道所有类型为止。这对最初加载的类

型来说，叫作“传递闭包”。[1]

下面看个简单的示例，我们来分析一下 `java.lang.Object` 类的依赖。图 11-1 显示的是简化的 `Object` 类依赖图，只显示了 `Object` 公开 API 可见的直接依赖，以及各个依赖的 API 可见的直接依赖。而且，反射子系统中 `Class` 类的依赖，以及 I/O 子系统中 `PrintStream` 和 `PrintWriter` 类的依赖也做了大量简化。

从图 11-1 可以看出 `Object` 类的部分传递闭包。

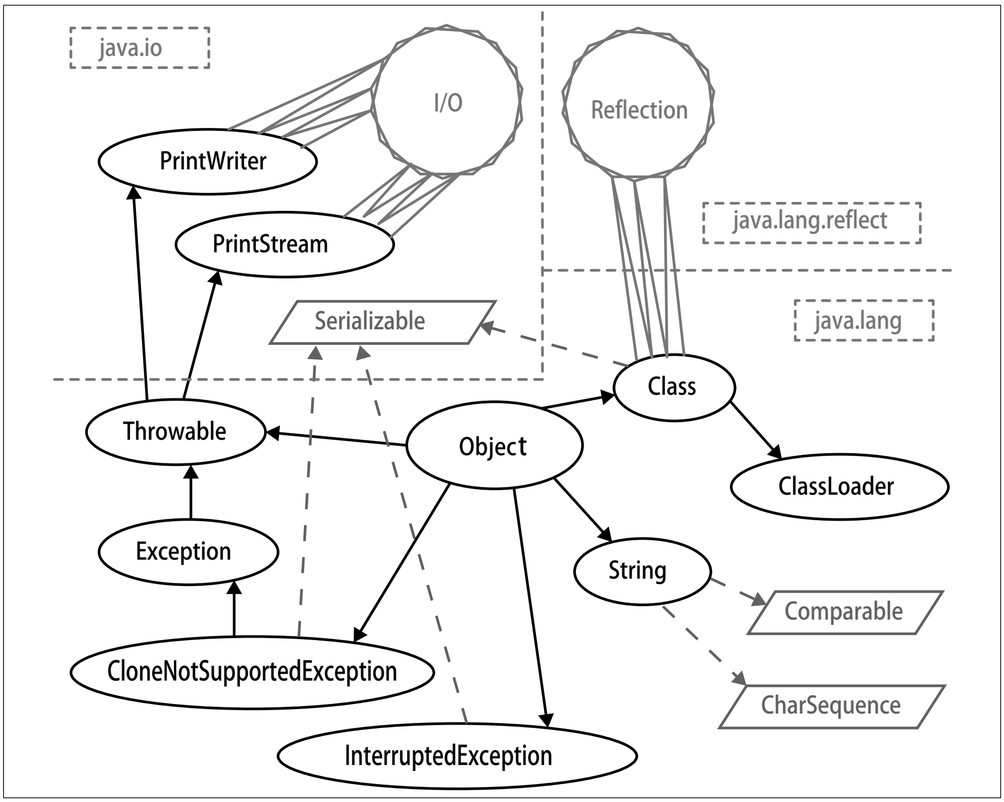


图 11-1：类型的传递闭包

11.2.4 初始化

解析阶段结束后，JVM 终于可以初始化类了。这个阶段会初始化静态变量，还会运行静态初始化代码块。

这是 JVM 首次执行新加载的类的字节码。静态初始化代码块运行完毕后，类就完全加载

注 1：和第 6 章一样，我们从数学的图论分支中借用了“传递闭包”这种说法。

好，可以使用了。

11.3　安全的编程和类加载

Java 程序能从多种源动态加载 Java 类，包括不信任的源，例如能通过不安全的网络访问的网站。动态创建和使用这种动态代码源是 Java 的一大优势和特性。不过，为了让这种机制能正常运作，Java 着重强调了一种安全架构，让不信任的代码能安全运行，而不用担心会损害宿主系统。

Java 的类加载子系统实现了很多安全功能。类加载架构的核心安全机制是，只允许使用一种方式把可执行的代码传入进程——类。

由此我们看到了希望，因为创建新类只有一种方式，即使用 `Classloader` 类提供的功能，从字节流中加载类。所以，我们可以限制需要保护的攻击面，全力保障类加载过程的安全性。

JVM 的一个特性对此特别有帮助：JVM 是栈机器。因此，所有操作都在栈中执行，而不在寄存器中执行。栈的状态在方法内的任何地方都能推知，这一点可以保证字节码不会破坏安全模型。

JVM 实现的一些安全检查措施如下所示：

- 类的所有字节码都有有效的参数；
- 调用所有方法时，传入的参数数量都正确，而且静态类型也正确；
- 字节码决不能试图上溢或下溢 JVM 栈；
- 局部变量在初始化之前不能使用；
- 只能把类型合适的值赋值给变量；
- 必须考虑字段、方法和类的访问控制修饰符；
- 没有危险的校正（例如，试图把 `int` 类型的值校正为指针）；
- 所有分支指令都指向同一个方法中的有效位置。

其中最重要的是对内存和指针的检查。在汇编语言和 C/C++ 中，整数和指针可以相互转换，所以整数能用作内存地址。使用汇编语言可以编写如下代码：

```
mov eax, [STAT] ; 从STAT地址中移动4个字节到eax中
```

Java 安全架构的最底层涉及 Java 虚拟机和它执行的字节码的设计方式。JVM 不允许使用任何方式直接访问底层系统中的单个内存地址，因此 Java 代码无法干扰本地硬件和操作系统。这些特意为 JVM 制定的限制在 Java 语言中也有体现，即 Java 不支持指针或指针运算。

Java 语言和 JVM 都不允许把整数校正成对象引用，反之亦然。而且，无论如何都不能获取对象在内存中的地址。没有这种功能，恶意代码就没了立足之地。

第 2 章说过，Java 的值有两种类型——基本类型和对象引用。只有这两种值能赋值给变量。注意，“对象的内容”不能赋值给变量。Java 没有 C 语言中的结构体（`struct`），而且传递的始终是值。对引用类型来说，传递的是引用副本——这也是值。

引用在 JVM 中使用指针表示，不过，指针不直接通过字节码处理。事实上，字节码没有用于访问特定位置内存的操作码。

我们所能做的只是访问字段和方法，字节码不能调用任意位置的内存。这意味着，JVM 始终知道代码和数据之间的区别。因此，这样能避免一整类栈溢出和其他攻击。

11.4 应用类加载知识

若想应用类加载知识，一定要完全理解 `java.lang.ClassLoader` 类。

这是个抽象类，功能完善，没有抽象方法。之所以使用 `abstract` 修饰符是为了强调，若想使用，必须创建子类。

除了前面提到的 `defineClass()` 方法，还可以使用公开的 `loadClass()` 方法加载类。子类 `URLClassLoader` 一般会使用这个方法，从 URL 或文件路径中加载类。

我们可以使用 `URLClassLoader` 对象从本地硬盘中加载类，如下所示：

```
String current = new File( "." ).getCanonicalPath();
try (URLClassLoader ulr =
  new URLClassLoader(new URL[] {new URL("file://"+ current + "/")})) {
  Class<?> clz = ulr.loadClass("com.example.DFACaller");
  System.out.println(clz.getName());
}
```

`loadClass()` 方法的参数是类文件的二进制名。注意，类文件必须存放在文件系统中的预定位置，`URLClassLoader` 对象才能找到指定的类。例如，要在相对于工作目录的 com/example/DFACaller.class 文件中才能找到 `com.example.DFACaller` 类。

`Class` 类还提供了 `Class.forName()` 方法，这是个静态方法，能从类路径中加载还未被引用的类。

这个方法的参数是类的完全限定名称。例如：

```
Class<?> jdbcClz = Class.forName("oracle.jdbc.driver.OracleDriver");
```

如果找不到指定的类，这个方法会抛出 `ClassNotFoundException` 异常。如这个示例所示，`forName()` 方法在旧版 JDBC 中经常使用，目的是确保加载了正确的驱动器。如果使用

`import`，会把依赖导入使用驱动器的类，`forName()` 方法则能避免这个问题。

JDBC 4.0 之后，不再需要这个初始化步骤了。

`Class.forName()` 方法还有一种形式，接受三个参数，有时会和另一个类加载程序一起使用：

```
Class.forName(String name, boolean inited, Classloader classloader);
```

`ClassLoader` 类有很多子类，分别处理各种特殊的类加载过程。这些子类组成了类加载程序层次结构。

类加载程序层次结构

JVM 有多个类加载程序，而且形成一个层次结构，每个类加载程序（除了第一层“原始”类加载程序）都可以把工作交给父级类加载程序完成。

按照约定，类加载程序会要求父级类加载程序解析并加载类，只有父级类加载程序无法完成时才会自己动手。一些常用的类加载程序如图 11-2 所示。

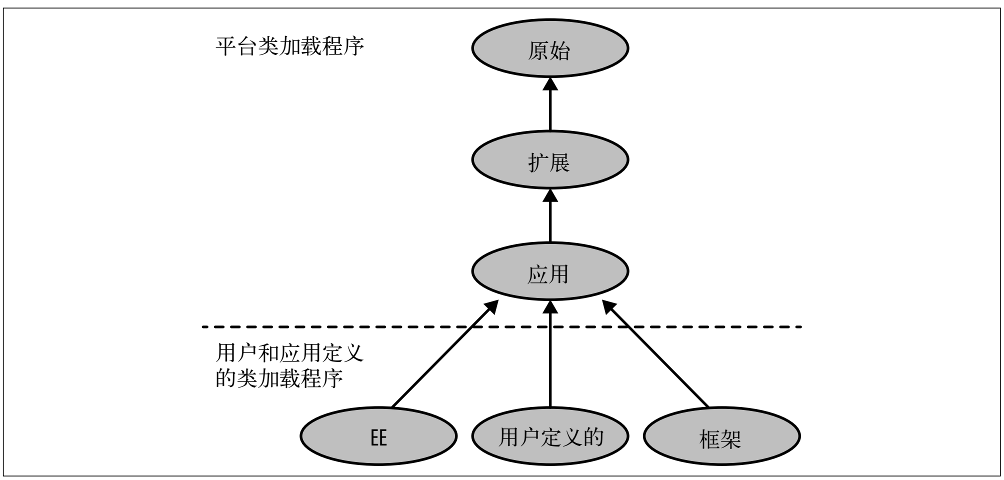


图 11-2：类加载程序的层次结构

1. 原始类加载程序

这是所有 JVM 进程中出现的第一个类加载程序，只用来加载核心系统类（在 rt.jar 中）。这个类加载程序不做验证，安全性靠引导类路径（boot classpath）保障。

引导类路径可以使用 `-Xbootclasspath` 选项调整，详情参见第 13 章。

2. 扩展类加载程序

这个类加载程序只用于加载 JDK 扩展——扩展一般保存在 JVM 安装目录中的 lib/ext 目录里。

扩展类加载程序的父级类加载程序是原始类加载程序。这个类加载程序使用不广泛，不过有时会用来实现调试器及相关的开发工具。

Nashorn JavaScript 环境（参见第 12 章）也使用这个类加载程序加载。

3. 应用类加载程序

这个类加载程序以前叫系统类加载程序，这个名称可不好，因为它并不加载系统（这是原始类加载程序的工作）。应用类加载程序的作用是从类路径中加载应用代码。这个类加载程序最常见，其父级类加载程序是扩展类加载程序。

应用类加载程序使用非常广泛，但很多高级 Java 框架需要的功能，这些主要的类加载程序没有提供，因此要扩展标准的类加载程序。“自定义类加载”的基础是，实现 `ClassLoader` 的新子类。

4. 自定义类加载程序

加载类时，迟早要把数据变成代码。前面说过，`defineClass()` 方法（其实是一组相关的方法）的作用是把 `byte[]` 数组转换成类对象。

这个方法通常在子类中调用。例如，下面这个简单的自定义类加载程序从硬盘中读取文件，创建类对象：

```
public static class DiskLoader extends ClassLoader {
  public DiskLoader() {
    super(DiskLoader.class.getClassLoader());
  }

  public Class<?> loadFromDisk(String clzName) throws IOException {
    byte[] b = Files.readAllBytes(Paths.get(clzName));

    return defineClass(null, b, 0, b.length);
  }
}
```

注意，上述示例和 `URLClassLoader` 类的示例不同，不用把类文件存放在硬盘中“正确的”位置。

每个自定义类加载程序都要有父级类加载程序。在这个示例中，我们把加载 `DiskLoader` 类的类加载程序（通常都是应用类加载程序）指定为它的父级类加载程序。

自定义类加载这个技术在 Java EE 和高级的 SE 环境中十分常见，目的是为 Java 平台提供非常复杂的功能。本章后面会举个自定义类加载的例子。

动态类加载有个缺点：使用动态加载的类对象时，往往对这个类知之甚少或一无所知。为了有效使用这个类，我们通常要使用一套动态编程技术——反射。

11.5 反射

反射是在运行时审查、操作和修改对象的能力，可以修改对象的结构和行为，甚至还能自我修改。

即便编译时不知道类型和方法名称，也能使用反射。反射使用类对象提供的基本元数据，能从类对象中找出方法或字段的名称，然后获取表示方法或字段的对象。

（使用 `Class::newInstance()` 或另一个构造方法）创建实例时也能让实例具有反射功能。如果有一个能反射的对象和一个 `Method` 对象，我们就能在之前类型未知的对象上调用任何方法。

因此，反射是一种十分强大的技术，所以，我们要知道什么时候可以使用，什么时候由于功能太强而不能使用。

11.5.1 什么时候使用反射

很多，也许是多数 Java 框架都会适度使用反射。如果编写的架构足够灵活，在运行时之前都不知道要处理什么代码，那么通常都需要使用反射。例如，插入式架构、调试器、代码浏览器和 REPL 类环境往往都会在反射的基础上实现。

反射在测试中也有广泛应用，例如，JUnit 和 TestNG 库都用到了反射，而且创建模拟对象也要使用反射。如果你用过任何一个 Java 框架，即便没有意识到，也几乎可以确定，你使用的是具有反射功能的代码。

在自己的代码中使用反射 API 时一定要知道，获取到的对象几乎所有信息都未知，因此处理起来可能很麻烦。

只要知道动态加载的类的一些静态信息（例如，加载的类实现一个已知的接口），与这个类交互的过程就能大大简化，减轻反射操作的负担。

使用反射时有个常见的误区：试图创建能适用于所有场合的反射框架。正确的做法是，只处理当前领域立即就能解决的问题。

11.5.2 如何使用反射

任何反射操作的第一步都是获取一个 `Class` 对象，表示要处理的类型。有了这个对象，就能访问表示字段、方法或构造方法的对象，并将其应用于未知类型的实例。

获取未知类型的实例，最简单的方式是使用没有参数的构造方法，这个构造方法可以直接在 `Class` 对象上调用：

```
Class<?> clz = getSomeClassObject();
Object rcvr = clz.newInstance();
```

如果构造方法有参数，必须找到具体需要使用的构造方法，并使用 Constructor 对象表示。

Method 对象是反射 API 提供的对象中最常使用的，下面会详细讨论。Constructor 和 Field 对象在很多方面都和 Method 对象类似。

1. Method对象

类对象中包含该类中每个方法的 Method 对象。这些 Method 对象在类加载之后惰性创建，所以在 IDE 的调试器中不会立即出现。

我们看一下 Method 类的源码，看看 Method 对象中保存了方法的哪些信息和元数据：

```
private Class<?>                   clazz;
private int                        slot;
// This is guaranteed to be interned by the VM in the 1.4
// reflection implementation
private String                     name;
private Class<?>                   returnType;
private Class<?>[]                 parameterTypes;
private Class<?>[]                 exceptionTypes
private int                        modifiers;
// Generics and annotations support
private transient String           signature;
// Generic info repository; lazily initialized
private transient MethodRepository genericInfo;
private byte[]                     annotations;
private byte[]                     parameterAnnotations;
private byte[]                     annotationDefault;
private volatile MethodAccessor    methodAccessor;
```

Method 对象提供了所有可用信息，包括方法能抛出的异常和注解（保留 RUNTIME 异常的策略），甚至还有会被 javac 移除的泛型信息。

Method 对象中的元数据可以调用访问器方法查看，不过一直以来，Method 对象的最大用处是反射调用。

这些对象表示的方法可以在 Method 对象上使用 invoke() 方法调用。下面这个示例在 String 对象上调用 hashCode() 方法：

```
Object rcvr = "a";
try {
  Class<?>[] argTypes = new Class[] { };
  Object[] args = null;

  Method meth = rcvr.getClass().getMethod("hashCode", argTypes);
  Object ret = meth.invoke(rcvr, args);
  System.out.println(ret);
```

```
} catch (IllegalArgumentException | NoSuchMethodException |
        SecurityException e) {
  e.printStackTrace();
} catch (IllegalAccessException | InvocationTargetException x) {
  x.printStackTrace();
}
```

为了获取想使用的 Method 对象，我们在类对象上调用 getMethod() 方法，得到的是一个 Method 对象的引用，指向这个类中对应的公开方法。

注意，变量 rcvr 的静态类型是 Object。在反射调用的过程中不会用到静态类型信息。invoke() 方法返回的也是 Object 对象，所以 hashCode() 方法真正的返回值被自动打包成了 Integer 类型。

从自动打包可以看出，反射 API 有些方面稍微有点难处理——下一节详述。

2. 反射的问题

Java 的反射 API 往往是处理动态加载代码的唯一方式，不过 API 中有些让人头疼的地方，处理起来稍微有点困难：

- 大量使用 Object[] 表示调用参数和其他实例；
- 大量使用 Class[] 表示类型；
- 同名方法可以重载，所以需要维护一个类型组成的数组，区分不同的方法；
- 不能很好地表示基本类型——需要手动打包和拆包。

void 就是个明显的问题——虽然有 void.class，但没坚持用下去。Java 甚至不知道 void 是不是一种类型，而且反射 API 中的某些方法使用 null 代替 void。

这很难处理，而且容易出错，尤其是稍微有点冗长的数组句法，更容易出错。

处理非公开方法的方式是更大的问题。我们不能使用 getMethod() 方法，必须使用 getDeclaredMethod() 方法才能获取非公开方法的引用，而且还要使用 setAccessible() 方法覆盖 Java 的访问控制子系统，然后才能执行非公开方法：

```
public class MyCache {
  private void flush() {
    // 清除缓存……
  }
}

Class<?> clz = MyCache.class;
try {
  Object rcvr = clz.newInstance();
  Class<?>[] argTypes = new Class[] { };
  Object[] args = null;
```

```
        Method meth = clz.getDeclaredMethod("flush", argTypes);
        meth.setAccessible(true);
        meth.invoke(rcvr, args);
    } catch (IllegalArgumentException | NoSuchMethodException |
             InstantiationException | SecurityException e) {
        e.printStackTrace();
    } catch (IllegalAccessException | InvocationTargetException x) {
        x.printStackTrace();
    }
```

不过，需要指出的是，使用反射的过程中始终会涉及未知信息。从某种程度上看，为了能处理反射调用，为了能使用反射 API 为开发者提供的运行时动态功能，我们只能容忍这种啰嗦的方式。

下面是本节最后一个示例。这个示例把反射和自定义类加载结合在一起使用，检查硬盘中的类文件里是否包含弃用方法（弃用方法应该使用 `@Deprecated` 标记）：

```
public class CustomClassloadingExamples {
    public static class DiskLoader extends ClassLoader {

        public DiskLoader() {
            super(DiskLoader.class.getClassLoader());
        }

        public Class<?> loadFromDisk(String clzName)
          throws IOException {
            byte[] b = Files.readAllBytes(Paths.get(clzName));

            return defineClass(null, b, 0, b.length);
        }
    }

    public void findDeprecatedMethods(Class<?> clz) {
        for (Method m : clz.getMethods()) {
            for (Annotation a : m.getAnnotations()) {
                if (a.annotationType() == Deprecated.class) {
                    System.out.println(m.getName());
                }
            }
        }
    }

    public static void main(String[] args)
      throws IOException, ClassNotFoundException {
        CustomClassloadingExamples rfx =
          new CustomClassloadingExamples();

        if (args.length > 0) {
            DiskLoader dlr = new DiskLoader();
            Class<?> clzToTest = dlr.loadFromDisk(args[0]);
            rfx.findDeprecatedMethods(clzToTest);
        }
```

```
    }
}
```

11.6 动态代理

Java 的反射 API 还有最后一个功能没讲——创建动态代理。动态代理是实现了一些接口的类（扩展 `java.lang.reflect.Proxy` 类）。这些类在运行时动态创建，而且会把所有调用都转交给 `InvocationHandler` 对象处理：

```
InvocationHandler h = new InvocationHandler() {
  @Override
  public Object invoke(Object proxy, Method method, Object[] args)
                                      throws Throwable {
    String name = method.getName();
    System.out.println("Called as: "+ name);
    switch (name) {
      case "isOpen":
        return false;
      case "close":
        return null;
    }

    return null;
  }
};

Channel c =
  (Channel) Proxy.newProxyInstance(Channel.class.getClassLoader(),
                          new Class[] { Channel.class }, h);

c.isOpen();
c.close();
```

代理可以用作测试的替身对象（尤其是测试使用模拟方式实现的对象）。

代理的另一个作用是提供接口的部分实现，或者修饰或控制委托对象的某些方面：

```
public class RememberingList implements InvocationHandler {
  private final List<String> proxied = new ArrayList<>();

  @Override
  public Object invoke(Object proxy, Method method, Object[] args)
                       throws Throwable {
    String name = method.getName();
    switch (name) {
      case "clear":
        return null;
      case "remove":
      case "removeAll":
        return false;
    }
```

```
        return method.invoke(proxied, args);
      }
    }

    RememberingList hList = new RememberingList();

    List<String> l =
      (List<String>) Proxy.newProxyInstance(List.class.getClassLoader(),
                                            new Class[] { List.class },
                                            hList);
    l.add("cat");
    l.add("bunny");
    l.clear();
    System.out.println(l);
```

代理的功能特别强大而且灵活，很多 Java 框架都有使用。

11.7　方法句柄

Java 7 引入了全新的内省和方法访问机制。这种机制原本是为动态语言设计的，运行时可能需要加入方法调度决策机制。为了在 JVM 层支持这个机制，Java 引入了一个新字节码——`invokedynamic`。Java 7 并没有使用这个字节码，在 Java 8 中才大量用于 lambda 表达式和 Nashorn JavaScript 引擎中。

即便没有 `invokedynamic`，新方法句柄 API 的功能在很多方面也和反射 API 差不多，而且用起来更简洁，提出的概念更简单，就算单独使用也没问题。方法句柄可以理解成安全且现代化的反射。

11.7.1　MethodType对象

在反射 API 中，方法签名使用 `Class[]` 表示，这样处理起来很麻烦。而方法句柄 API 则使用 `MethodType` 对象表示。使用这种方式表示方法的类型签名更安全，而且更符合面向对象思想。

`MethodType` 对象包含返回值的类型和参数类型，但没有接收者的类型或方法的名称。因为没有方法的名称，所以具有正确签名的方法可以绑定到任何名称上（参照 lambda 表达式的函数式接口行为）。

方法的类型签名通过工厂方法 `MethodType.methodType()` 获取，是 `MethodType` 类的实例，而且不可变。例如：

```
MethodType m2Str = MethodType.methodType(String.class); // toString()

// Integer.parseInt()
```

```
MethodType mtParseInt =
  MethodType.methodType(Integer.class, String.class);

// ClassLoader类中的defineClass()方法
MethodType mtdefClz = MethodType.methodType(Class.class, String.class,
                                            byte[].class, int.class,
                                            int.class);
```

虽然获取 MethodType 对象的方式看起来让人困惑，但获得的效果比反射 API 好，因为表示和讨论方法签名都更容易。下一步是通过一个查找过程，获取方法的句柄。

11.7.2 方法查找

方法查找查询在定义方法的类中执行，而且结果取决于执行查询的上下文。从下述示例可以看出，在一般的上下文中试图查找受保护的 Class::defineClass() 方法会失败，抛出 IllegalAccessException 异常，因为无法访问这个受保护的方法：

```
public static void lookupDefineClass(Lookup l) {
  MethodType mt = MethodType.methodType(Class.class, String.class,
                                        byte[].class, int.class,
                                        int.class);

  try {
    MethodHandle mh =
      l.findVirtual(ClassLoader.class, "defineClass", mt);
    System.out.println(mh);
  } catch (NoSuchMethodException | IllegalAccessException e) {
    e.printStackTrace();
  }
}

Lookup l = MethodHandles.lookup();
lookupDefineClass(l);
```

我们一定要调用 MethodHandles.lookup() 方法，基于当前执行的方法获取上下文 Lookup 对象。

在 Lookup 对象上可以调用几个方法（方法名都以 find 开头），查找需要的方法，包括 findVirtual()、findConstructor() 和 findStatic()。

反射 API 和方法句柄 API 之间一个重大的区别是处理访问控制的方式。Lookup 对象只会返回在创建这个对象的上下文中可以访问的方法——没有任何方式能破坏这个规则（不像反射 API 可以使用 setAccessible() 方法调整访问控制）。

因此，方法句柄始终会遵守安全规则，而使用反射 API 的等效代码可能做不到这一点。方法句柄会在构建查找上下文时检查访问权限，所以不会为没有正确访问权限的方法创建句柄。

Lookup 对象或从中获取的方法句柄，可以返回给其他上下文，包括不再能访问该方法的上下文。在这种情况下，句柄依然是可以执行的，因为访问控制在查询时检查。这一点从下面的示例可以看出：

```
public class SneakyLoader extends ClassLoader {
  public SneakyLoader() {
    super(SneakyLoader.class.getClassLoader());
  }

  public Lookup getLookup() {
    return MethodHandles.lookup();
  }
}

SneakyLoader snLdr = new SneakyLoader();
l = snLdr.getLookup();
lookupDefineClass(l);
```

通过 Lookup 对象可以为任何能访问的方法生成方法句柄，还能访问方法无法访问的字段。在 Lookup 对象上调用 findGetter() 和 findSetter() 方法，分别可以生成读取字段和更新字段的方法句柄。

11.7.3 调用方法句柄

方法句柄表示调用方法的能力。方法句柄对象是强类型的，会尽量保证类型安全。方法句柄都是 java.lang.invoke.MethodHandle 类的子类实例，JVM 会使用特殊的方式处理这个类。

调用方法句柄有两种方式——使用 invoke() 方法或 invokeExact() 方法。这两个方法的参数都是接收者和调用参数。invokeExact() 方法尝试直接调用方法句柄，而 invoke() 方法在需要时会修改调用参数。

一般来说，invoke() 方法会调用 asType() 方法转换参数。转换的规则如下。

- 如果需要，打包基本类型的参数。
- 如果需要，拆包打包好的基本类型参数。
- 如果需要，放大转换基本类型的参数。
- 会把 void 返回类型修改为 0 或 null，具体是哪个取决于期待的返回值是基本类型还是引用类型。
- 不管静态类型是什么，都能传入 null。

考虑到可能会执行这些转换，所以要像下面这样调用方法句柄：

```
Object rcvr = "a";
try {
  MethodType mt = MethodType.methodType(int.class);
```

```
  MethodHandles.Lookup l = MethodHandles.lookup();
  MethodHandle mh = l.findVirtual(rcvr.getClass(), "hashCode", mt);

  int ret;
  try {
    ret = (int)mh.invoke(rcvr);
    System.out.println(ret);
  } catch (Throwable t) {
    t.printStackTrace();
  }
} catch (IllegalArgumentException |
  NoSuchMethodException | SecurityException e) {
  e.printStackTrace();
} catch (IllegalAccessException x) {
  x.printStackTrace();
}
```

方法句柄提供的动态编程功能和反射一样，但处理方式更清晰明了。而且，方法句柄能在 JVM 的低层执行模型中很好地运转，因此，性能比反射好得多。

第12章

Nashorn

甲骨文在 Java 8 中加入了一个运行在 JVM 中的 JavaScript 新实现——Nashorn。Nashorn 的目的是取代前一个运行在 JVM 中的 JavaScript 项目，即 Rhino（Nashorn 是德文中的“rhino”[1]）。

Nashorn 是完全重写的实现，争取做到能和 Java 轻易互通，获得高性能，并且严格遵守 JavaScript ECMA 规范。Nashorn 是首个 100% 遵守规范的 JavaScript 实现，而且在多数应用场合至少比 Rhino 快 20 倍。

12.1　介绍Nashorn

本章假定你对 JavaScript 有一定的了解。如果你还不熟悉 JavaScript 的基本概念，Michael Morrison 写的 *Head First JavaScript*（O'Reilly 出版）是本不错的入门书。

回忆一下 1.5.4 节介绍的 Java 和 JavaScript 之间的区别，你会发现，这两种语言十分不同。因此，能在 Java 的虚拟机中运行 JavaScript 看起来有点奇怪。

12.1.1　在JVM中运行Java之外的语言

其实，除了 Java 之外有相当多的语言都运行在 JVM 中，而且其中有些与 Java 的差异大于 JavaScript 与 Java 的差异。之所以能在 JVM 中运行其他语言，是因为 Java 语言和 JVM 耦合得非常松，两者之间只通过特定格式的类文件交互。在 JVM 中运行其他语言有

注 1：rhino 的意思是犀牛。——译者注

两种方式。

- 目标语言使用 Java 实现的解释器。
  解释器在 JVM 中运行，执行使用目标语言编写的程序。

- 目标语言提供编译器，把目标语言代码转换成类文件。
  编译得到的类文件直接在 JVM 中执行，通常还会提供一些语言专用的运行时功能。

Nashorn 采用的是第二种方式，不过做了改进，把编译器嵌入运行时，所以，执行程序前绝不会先编译 JavaScript 源代码。这意味着，不是专为 Nashorn 编写的 JavaScript 代码也能轻易部署到这个平台中。

> Nashorn 和很多其他运行在 JVM 中的语言（例如 JRuby）有个区别，它没有实现任何解释器。Nashorn 总是把 JavaScript 代码编译成 JVM 字节码，然后直接执行字节码。

从技术的角度来看，这种实现方式很有趣。不过很多开发者好奇的是，Nashorn 在已经建立好的 Java 成熟生态系统中扮演着怎样的角色。下面就来看看 Nashorn 扮演的角色。

12.1.2　目的

在 Java 和 JVM 生态系统中，Nashorn 有多种用途。首先，它为 JavaScript 开发者提供了一个可用的环境，用于探索 JVM 的功能。其次，它让企业继续利用对 Java 技术的现有投资，采用 JavaScript 作为一门开发语言。最后，它为 HotSpot 使用的先进虚拟机技术提供了一个很好的工程样板。

JavaScript 不断发展，应用范围越来越宽，以前只能在浏览器中使用，而现在则能在更通用的计算和服务器端使用。Nashorn 在稳固的 Java 现有生态系统和一波有前途的新技术之间架起了一座桥梁。

下面我们要介绍 Nashorn 运作的技术细节，以及如何开始使用这个平台。在 Nashorn 中运行 JavaScript 代码有多种不同的方式，下一节会介绍其中最常使用的两种。

12.2　在Nashorn中执行JavaScript代码

本节介绍 Nashorn 环境，还会讨论两种执行 JavaScript 代码的方式（这两种方式使用的工具都在 $JAVA_HOME 的子目录 bin 中）。

- `jrunscript`
  这是一个简单的脚本运行程序，执行 .js 格式的 JavaScript 文件。

- jjs

  这是一个功能更完整的 shell，既能运行脚本，也能作为交互式读取－求值－输出循环（Read-Eval-Print-Loop，REPL）环境使用，用于探索 Nashorn 及其功能。

我们先介绍基本的运行程序 jrunscript，它适用于大多数简单的 JavaScript 应用。

12.2.1 在命令行中运行

若想在 Nashorn 中执行名为 my_script.js 的 JavaScript 文件，使用 jrunscript 命令即可：

```
jrunscript my_script.js
```

除了 Nashorn，jrunscript 还能使用其他脚本引擎（12.3 节会详细介绍脚本引擎）。如果需要使用其他引擎，可以通过 -l 选项指定：

```
jrunscript -l nashorn my_script.js
```

> 如果有合适的脚本引擎，使用这个选项还能让 jrunscript 运行使用其他语言编写的脚本。

这个基本的运行程序特别适合在简单的应用场景中使用，不过它有一定的局限性，因此，在重要场合下，我们需要使用功能更强的执行环境——jjs，也就是 Nashorn shell。

12.2.2 使用Nashorn shell

启动 Nashorn shell 的命令是 jjs。Nashorn shell 既可以交互式使用，也可以非交互式使用，能直接替代 jrunscript。

最简单的 JavaScript 示例当然是经典的“Hello World”，我们看一下如何在交互式 shell 中编写这个示例：

```
$ jjs
jjs> print("Hello World!");
Hello World!
jjs>
```

在 shell 中可以轻易处理 Nashorn 和 Java 之间的相互操作，12.4.1 节会详细介绍，不过现在先举个例子。若想在 JavaScript 中直接访问 Java 类和方法，使用完全限定的类名即可。下面这个实例获取 Java 原生的正则表达式功能：

```
jjs> var pattern = java.util.regex.Pattern.compile("\\d+");
```

```
jjs> var myNums = pattern.split("a1b2c3d4e5f6");

jjs> print(myNums);
[Ljava.lang.String;@10b48321

jjs> print(myNums[0]);
a
```

在 REPL 中打印 JavaScript 变量 `myNums` 时，得到的结果是 `[Ljava.lang.String;@10b48321`。这表明，虽然 `myNums` 是 JavaScript 代码中的变量，但其实它是 Java 中的字符串数组。

稍后会详细说明 Nashorn 和 Java 之间的相互操作，现在先介绍 `jjs` 的其他功能。`jjs` 命令的通用格式是：

```
jjs [<options>] <files> [-- <arguments>]
```

能传给 `jjs` 的选项有很多，其中最常使用的如下所示。

- `-cp` 或 `-classpath`：指定在哪个位置寻找额外的 Java 类（稍后会发现，通过 `Java.type` 机制实现）。
- `-doe` 或 `-dump-on-error`：如果强制退出 Nashorn，转储完整的错误信息。
- `-J`：把选项传给 JVM。例如，如果想增加 JVM 可用的最大内存，可以这么做：

```
$ jjs -J-Xmx4g
jjs> java.lang.Runtime.getRuntime().maxMemory()
3817799680
```

- `-strict`：在 JavaScript 严格模式中执行所有脚本和函数。这是 ECMAScript 5 引入的 JavaScript 特性，目的是减少缺陷和错误。所有新编写的 JavaScript 代码都推荐使用严格模式，如果你对这个特性还不了解，应该找些资料看看。
- `-D`：让开发者把键值对表示的系统属性传给 Nashorn，这和 JVM 的通常做法一样。例如：

```
$ jjs -DmyKey=myValue
jjs> java.lang.System.getProperty("myKey");
myValue
```

- `-v` 或 `-version`：打印标准的 Nashorn 版本字符串。
- `-fv` 或 `-fullversion`：打印完整的 Nashorn 版本字符串。
- `-fx`：把脚本当成 JavaFX GUI 应用执行。JavaFX 程序员使用 Nashorn 可以少编写一些样板代码。[2]

注 2：JavaFX 是开发 GUI 应用的标准 Java 技术，不过超出了本书范畴。

- `-h`：显示帮助信息。
- `-scripting`：启用 Nashorn 专用的脚本扩展。下一节会介绍这个功能。

12.2.3 在jjs中编写脚本

`jjs` shell 可用于测试一些基本的 JavaScript 代码，或者使用交互式方式试验不熟悉的 JavaScript 包（例如，学习使用包时）。不过，`jjs` 有个限制，不能输入多行代码，也没提供其他大量使用 REPL 的语言常用的高级功能。

其实，`jjs` 非常适合在非交互式场合下使用，例如启动使用 JavaScript 编写的守护进程。对于这种情况，可以使用下述方式调用 `jjs`：

```
$ jjs -scripting my_script.js
```

这样调用，可以使用增强的 `jjs` 功能，其中包含一些有用的扩展。很多扩展都能让脚本程序员通过更熟悉的方式使用 Nashorn。

1. 脚本中的注释

在传统的 Unix 脚本中，`#` 符号表示注释，一直到行尾结束。而 JavaScript 使用 C/C++ 风格的注释，使用 // 表示注释，一直到行尾结束。Nashorn 支持这种注释方式，不过在脚本模式中也能使用 Unix 脚本的注释方式，因此，下述代码完全合法：

```
#!/usr/bin/jjs

# 在脚本模式中,这样写注释完全合法

print("After the comment");
```

2. 行内执行命令

资深 Unix 程序员通常把这个功能称为“反引号”。在 bash 脚本中，我们可以编写如下的代码，使用 Unix 的 `curl` 命令下载谷歌首页的内容：

```
echo "Google says: " `curl http://www.google.co.uk`
```

类似地，在 Nashorn 脚本中，我们也可以使用反引号（`）执行 Unix shell 命令。如下所示：

```
print("Google says: "+ `curl http://www.google.co.uk`);
```

3. 字符串插值

字符串插值是一种特殊的句法，无需连接字符串，就能直接插入变量的内容。在 Nashorn 脚本中，我们可以使用 `${<variable name>}` 把变量的值插入字符串。例如，前面下载网页内容的示例，使用插值后可以改写成：

```
var url = "www.google.co.uk";
```

```
var pageContents = `curl http://${url}`;

print("Google says: ${pageContents}");
```

4. 特殊变量

Nashorn 还提供了几个特殊的全局变量和函数，编写脚本时特别有用，而且普通的 JavaScript 中没有。例如，传入脚本的参数可以通过 `$ARG` 变量获取。参数必须使用约定的 -- 方式传入，像下面这样：

```
jjs test1.jjs -- aa bbb cccc
```

获取参数的方式如下所示：

```
print($ARG);

for(var i=0; i < $ARG.length; i++) {
    print("${i}: "+ $ARG[i]);
}
```

> `$ARG` 变量是一个 JavaScript 数组（观察传给 `print()` 方法后的表现可以看出来），而且要当成数组处理。学过其他语言的程序员可能觉得这种句法有点让人困惑，因为有些语言使用 `$` 符号表示标量变量。

我们能遇到的另一个特殊的全局变量是 `$ENV`，这个变量用于获取当前的环境变量。例如，下述代码打印当前用户的家目录：

```
print("HOME = "+ $ENV.HOME); # 在我的电脑中打印的是/home/ben
```

Nashorn 还提供了一个特殊的全局函数，`$EXEC()`。这个函数的作用和前面介绍的反引号类似，如下述示例所示：

```
var execOutput = $EXEC("echo Print this on stdout");
print(execOutput);
```

你可能注意到了，使用反引号或 `$EXEC()` 函数时，不会打印所执行命令的输出，而是返回函数的返回值。这是为了避免命令的输出扰乱主脚本的输出。

Nashorn 提供了另外两个特殊的变量，有助于程序员处理脚本中所执行命令的输出：`$OUT` 和 `$ERR`。这两个变量分别用于捕获脚本中所执行命令的输出和错误消息。例如：

```
$EXEC("echo Print this on stdout");

// 没有修改标准输出的代码

var saveOut = $OUT;
```

```
print("- - - - - - -");
print(saveOut);
```

$OUT 和 $ERR 中的内容一直存在，除非主脚本中的后续代码修改这些内容（例如执行其他命令）。

5. 行内文档

JavaScript 和 Java 一样，不支持把包围字符串的两个引号放在不同的行（这种字符串叫多行字符串）。可是，在脚本模式中，Nashorn 通过扩展对此提供了支持。这种功能也叫行内文档或 heredoc，是脚本语言的通用特性。

heredoc 以 <<END_TOKEN 开头，从下一行开始，直到结束符号（可以使用任何字符串，不过经常全部大写，经常使用的字符串有 END、END_DOC、END_STR、EOF 和 EOSTR），中间都是多行字符串的内容。在结束符号之后，脚本恢复正常。我们看一个示例：

```
var hw = "Hello World!";
var output = <<EOSTR;

This is a multiline string
It can interpolate too - ${hw}
EOSTR
print(output);
```

6. Nashorn提供的辅助函数

Nashorn 还提供了一些辅助函数，让开发者能轻易实现 shell 脚本经常要执行的常见任务。

- `print()/echo()`

  前面的示例都用到了 `print()` 函数。这两个函数的表现和预期完全一样，把传入的字符串打印出来，后面还会加一个换行符。

- `quit()/exit()`

  这两个函数的作用完全一样——退出脚本。可以把一个整数参数传给这两个函数，作为脚本进程的返回码。如果没传入参数，返回码默认为 0——这是 Unix 进程的习惯做法。

- `readLine()`

  从标准输入（通常是键盘）读取一行输入。默认情况下，这个函数会把读取的内容打印到标准输出，不过，如果把 `readLine()` 函数的返回值赋值给变量，输入的数据就在此结束，如下述示例所示：

  ```
  print("Please enter your name: ");
  var name = readLine();
  print("Please enter your age: ");
  var age = readLine();

  print(<<EOREC);
  ```

```
Student Record
-+-+-+-+-+-+-+-
Name: ${name}
Age:  ${age}
EOREC
```

- `readFully()`

  `readFully()` 函数不从标准输入读取数据，而是加载一个文件中的全部内容。和 `readLine()` 函数一样，加载的内容不是打印到标准输出，就是赋值给变量：

  ```
  var contents = readFully("input.txt");
  ```

- `load()`

  这个函数用于加载并执行脚本（使用 JavaScript 的 `eval` 函数执行）。脚本可以从本地路径或 URL 中加载。除此之外，还可以使用 JavaScript 的脚本对象表示法把脚本文件定义成一个字符串。

> 使用 `load()` 函数执行其他脚本可能出现意料之外的错误。JavaScript 支持使用 `try-catch` 块处理异常，所以加载代码时要使用这种方式。
>
> 下面举个简单的例子，这个例子在 Nashorn 中加载图形可视化库 D3:

```
try {
    load(“http://d3js.org/d3.v3.min.js”);
} catch (e) {
    print(“Something went wrong, probably that we’re not a web browser”);
}
```

- `loadWithNewGlobal()`

  `load()` 函数会在当前 JavaScript 上下文中执行加载的脚本。而有时我们想把代码放入属于它自己的纯净上下文中。此时，可以使用 `loadWithNewGlobal()` 函数，在全新的全局上下文中执行脚本。

7. shebang句法

本节介绍的所有功能都是为了方便编写 shell 脚本，让 `jjs` 能替代 bash、Perl 或其他脚本语言。为了完善这种支持，最后还需要一个功能——“shebang”句法，用来启动使用 Nashorn 编写的脚本。

> 如果可执行脚本的第一行以 `#!` 开头，而且后面是一个可执行文件的路径，那么 Unix 操作系统会假定这个路径指向一个解释器，而且这个解释器能处理这种脚本。执行脚本时，操作系统会启动指定的解释器，并把脚本文件传给解释器处理。

对 Nashorn 来说，最好创建一个符号链接（可能需要 sudo 访问），把 /usr/bin/jjs（或 /usr/local/bin/jjs）指向 jjs 的真正位置（通常是 $JAVA_HOME/bin/jjs）。然后，可以像下面这样编写 Nashorn shell 脚本：

```
#!/usr/bin/jjs

# ……脚本中的其他内容
```

对更高级的应用场景来说（例如长时间运行的守护进程），Nashorn 甚至提供了对 Node.js 的支持。这个功能由 Avatar 项目中的 Avatar.js 提供，详情参见 12.5 节的“Avatar 项目”。

本节介绍的工具便于直接在命令行中执行 JavaScript 代码，不过多数情况下，我们希望使用另一种方式执行 JavaScript 代码——在 Java 程序中调用 Nashorn，执行 JavaScript 代码。实现这种执行方式的 API 包含在 Java 包 javax.script 中，所以，接下来我们要介绍这个包，并讨论 Java 如何与解释脚本语言的引擎交互。

12.3 Nashorn和javax.script包

Nashorn 不是 Java 平台提供的第一个脚本语言。早在 Java 6 中就提供了 javax.script 包，这个包为引擎提供了通用接口，让脚本语言能和 Java 相互操作。

这个通用接口中包含脚本语言的基本概念，例如脚本代码的执行和编译方式（完整的脚本或者单个脚本语句是否在现有的上下文中）。而且还提出了脚本实体和 Java 绑定的概念，以及发现脚本引擎的功能。最后，javax.script 包提供了可选的调用功能（有别于执行，因为调用功能的作用是从脚本语言的运行时中导出中间代码，提供给 JVM 运行时使用）。

本节的示例使用 Rhino 语言编写，不过也有很多其他语言利用了 javax.script 包提供的功能。Java 8 移除了 Rhino，现在 Java 平台提供的默认脚本语言是 Nashorn。

通过javax.script包使用Nashorn

我们看一个非常简单的示例，这个示例展示了如何在 Java 代码中使用 Nashorn 运行 JavaScript 代码：

```
import javax.script.*;

ScriptEngineManager m = new ScriptEngineManager();
ScriptEngine e = m.getEngineByName("nashorn");

try {
  e.eval("print('Hello World!');");
} catch (final ScriptException se) {
  // ...
}
```

这段代码的重点是 ScriptEngine 对象，这个对象通过 ScriptEngineManager 对象获取。ScriptEngine 对象提供一个空脚本环境，然后调用 eval() 方法把代码添加到这个环境中。

Nashorn 引擎只提供了一个全局 JavaScript 对象，所以每次调用 eval() 方法都会在同一个环境中执行 JavaScript 代码。也就是说，我们可以多次调用 eval() 方法，在脚本引擎中逐渐积累 JavaScript 状态。例如：

```
e.eval("i = 27;");
e.put("j", 15);
e.eval("var z = i + j;");

System.out.println(((Number) e.get("z")).intValue()); // 打印42
```

注意，直接在 Java 代码中与脚本引擎交互有个问题：一般不知道值的任何类型信息。

然而，Nashorn 和 Java 的类型系统绑定得相当紧密，所以要稍微小心一些。在 Java 代码中使用 JavaScript 中等价的基本类型时，往往都会转换成对应的（包装）类型。例如，如果把下面这行代码添加到前一个示例的末尾：

```
System.out.println(e.get("z").getClass());
```

很明显，e.get("z") 获得的值是 java.lang.Integer 类型。如果稍微修改一下，改成：

```
e.eval("i = 27.1;");
e.put("j", 15);
e.eval("var z = i + j;");

System.out.println(e.get("z").getClass());
```

那么 e.get("z") 返回值的类型就变成了 java.lang.Double，由此体现了两种类型系统之间的区别。在其他 JavaScript 实现中，可能会把两种情况的返回值都当成数字类型（因为 JavaScript 没有定义整数类型）。可是，Nashorn 对数据的真正类型知道的更多。

> 使用 JavaScript 时，Java 程序员必须清醒地认识到，Java 的静态类型和 JavaScript 类型的动态本性之间是有区别的。如果没认识到这一点，很容易在不经意间引入缺陷。

在上述示例中，我们在 ScriptEngine 对象上调用 get() 和 put() 方法。这么做可以直接获取和设定 Nashorn 引擎当前全局作用域中的对象，无需直接编写或使用 eval() 方法执行 JavaScript 代码。

javax.script API

本节最后，我们要简介 javax.script API 中的关键类和接口。这个 API 相当小（6 个接口，5 个类，1 个异常），自从 Java 6 引入之后就没改动过。

- `ScriptEngineManager`类

  这个类是脚本功能的入口点，维护着一组当前进程中可用的脚本实现。这个功能由 Java 的服务提供者（service provider）机制实现，这种机制经常用于管理不同实现之间可能有较大差异的 Java 平台扩展。默认情况下，唯一可用的脚本扩展是 Nashorn，不过，也能使用其他脚本环境（例如 Groovy 和 JRuby）。

- `ScriptEngine`接口

  这个接口表示脚本引擎，作用是维护解释脚本的环境。

- `Bindings`接口

  这个接口扩展 `Map` 接口，把字符串（变量或其他符号的名称）映射到脚本对象上。Nashorn 使用这个接口实现 `ScriptObjectMirror` 机制，让 Java 和 JavaScript 代码能相互操作。

其实，多数应用只会使用 `ScriptEngine` 这个相对不透明的接口提供的方法，例如 `eval()`、`get()` 和 `put()` 方法，不过，理解这个接口在整个脚本 API 中的作用，会对你有所帮助。

12.4 Nashorn的高级用法

Nashorn 是个复杂的编程环境，也是个适合部署应用的稳健平台，而且能和 Java 相互操作。这一节我们介绍一些高级用法，把 Java 代码集成在 JavaScript 代码中，还会深入 Nashorn 的一些实现细节，说明实现这种集成的方式。

12.4.1 在Nashorn中调用Java代码

我们知道，每个 JavaScript 对象都会编译成某个 Java 类的实例，那么，你或许就不会奇怪，在 Nashorn 中能无缝集成 Java 代码——尽管二者的类型系统和语言特性有重要区别。不过，为了实现这种集成，还需要实现一些机制。

我们已经知道，在 Nashorn 中可以直接访问 Java 类和方法，例如：

```
$ jjs -Dkey=value
jjs> print(java.lang.System.getProperty("key"));
value
```

下面仔细分析一下这个句法，看看 Nashorn 是如何实现这种功能的。

1. JavaClass和JavaPackage类型

在 Java 中，表达式 `java.lang.System.getProperty("key")` 的意思是通过完全限定的名称调用 `java.lang.System` 类中的 `getProperty()` 静态方法。不过，在 JavaScript 的句法中，这个表达式的意思是，从符号 `java` 开始，链式访问属性。下面在 `jjs` shell 中看一下这些符号的表现：

```
jjs> print(java);
[JavaPackage java]

jjs> print(java.lang.System);
[JavaClass java.lang.System]
```

可以看出，java 是个特殊的 Nashorn 对象，用于访问 Java 系统中的包。在 JavaScript 中，Java 包使用 JavaPackage 类型表示，而 Java 类使用 JavaClass 类型表示。任何顶层包都能直接作为包导航对象，子包则可以赋值给 JavaScript 对象。因此，可以使用简短的句法访问 Java 类：

```
jjs> var juc = java.util.concurrent;
jjs> var chm = new juc.ConcurrentHashMap;
```

除了可以使用包对象导航之外，还可以使用另一个对象 Java。在这个对象上可以调用一些有用的方法，其中一个最重要的方法是 Java.type()。使用这个方法可以查询 Java 的类型系统，访问 Java 类。例如：

```
jjs> var clz = Java.type("java.lang.System");
jjs> print(clz);
[JavaClass java.lang.System]
```

如果在类路径（例如，使用 jjs 的 -cp 选项指定）中找不到指定的类，会抛出 ClassNotFoundException 异常（jjs 会把这个异常包装在一个 Java RuntimeException 异常对象中）：

```
jjs> var klz = Java.type("Java.lang.Zystem");
java.lang.RuntimeException: java.lang.ClassNotFoundException:
  Java.lang.Zystem
```

多数情况下，JavaScript 中的 JavaClass 对象都可以像 Java 的类对象一样使用（这两个类型稍微有所不同，不过可以把 JavaClass 理解为类对象在 Nashorn 中的镜像）。例如，在 Nashorn 中可以直接使用 JavaClass 创建 Java 新对象：

```
jjs> var clz = Java.type("java.lang.Object");
jjs> var obj = new clz;
jjs> print(obj);
java.lang.Object@73d4cc9e

jjs> print(obj.hashCode());
1943325854

// 注意,这种句法不起作用
jjs> var obj = clz.new;
jjs> print(obj);
undefined
```

不过，使用时要稍微小心一些。jjs 环境会自动打印表达式的结果，这可能会导致一些意

料之外的行为：

```
jjs> var clz = Java.type("java.lang.System");
jjs> clz.out.println("Baz!");
Baz!
null
```

这里的问题是，`java.lang.System.out.println()` 方法有返回值，类型为 `void`。而在 `jjs` 中，如果表达式没赋值给变量，就会得到一个值，并打印出来。所以，`println()` 方法的返回值会映射到 JavaScript 的 `null` 值上，并打印出来。

> 不熟悉 JavaScript 的 Java 程序员要注意，在 JavaScript 中处理 `null` 和缺失值很麻烦，尤其是 `null != undefined`。

2. JavaScript函数和Java lambda表达式

JavaScript 和 Java 之间的相互操作层级非常深，甚至可以使用 JavaScript 函数作为 Java 接口的匿名实现（或者作为 lambda 表达式）。下面举个例子，使用 JavaScript 函数作为 `Callable` 接口的实例（表示后续调用的代码块）。`Callable` 接口中只有一个方法，`call()`，这个方法没有参数，返回值是 `void`。在 Nashorn 中，我们可以使用 JavaScript 函数作为 lambda 表达式：

```
jjs> var clz = Java.type("java.util.concurrent.Callable");
jjs> print(clz);
[JavaClass java.util.concurrent.Callable]
jjs> var obj = new clz(function () { print("Foo"); } );
jjs> obj.call();
Foo
```

这个示例要表明的基本事实是，在 Nashorn 中，JavaScript 函数和 Java lambda 表达式之间没有区别。和在 Java 中一样，函数会被自动转换成相应类型的对象。下面看一下如何在 Java 线程池中使用 Java 的 `ExecutorService` 对象执行一些 JavaScript 代码：

```
jjs> var juc = java.util.concurrent;
jjs> var exc = juc.Executors.newSingleThreadExecutor();
jjs> var clbl = new juc.Callable(function (){
  \java.lang.Thread.sleep(10000); return 1; });
jjs> var fut = exc.submit(clbl);
jjs> fut.isDone();
false
jjs> fut.isDone();
true
```

与等效的 Java 代码相比（就算使用 Java 8 引入的 lambda 表达式），样板代码的减少量十分

惊人。不过，lambda 表达式的实现方式导致了一些限制。例如：

```
jjs> var fut=exc.submit(function (){\
java.lang.Thread.sleep(10000); return 1;});
java.lang.RuntimeException: java.lang.NoSuchMethodException: Can't
unambiguously select between fixed arity signatures
[(java.lang.Runnable), (java.util.concurrent.Callable)] of the method
java.util.concurrent.Executors.FinalizableDelegatedExecutorService
.submit for argument types
[jdk.nashorn.internal.objects.ScriptFunctionImpl]
```

这里的问题是，线程池中有一个重载的 `submit()` 方法。一个版本的参数是一个 `Callable` 对象，而另一个版本的参数是一个 `Runnable` 对象。可是，JavaScript 函数（作为 lambda 表达式时）既能转换成 `Callable` 对象，也能转换成 `Runnable` 对象。这就是错误消息中出现“unambiguously select”（明确选择）的原因。运行时能选择其中任何一个，但不能在二者之间作出抉择。

12.4.2 Nashorn对JavaScript语言所做的扩展

前面说过，Nashorn 是完全遵守 ECMAScript 5.1 标准（这是 JavaScript 的标准）的实现。不过，除此之外，Nashorn 还实现了一些 JavaScript 语言句法扩展，让开发者的生活更轻松。经常使用 JavaScript 的开发者应该会熟悉这些扩展，有相当一部分扩展实现的都是 Mozilla JavaScript 方言中的功能。下面介绍几个最常使用和最有用的扩展。

1. 遍历循环

标准的 JavaScript 没有提供等同于 Java 遍历循环的句法，不过 Nashorn 实现了 Mozilla 使用的 for each in 循环，如下所示：

```
var jsEngs = [ "Nashorn", "Rhino", "V8", "IonMonkey", "Nitro" ];
for each (js in jsEngs) {
    print(js);
}
```

2. 单表达式函数

Nashorn 还支持另一个小小的句法增强，目的是让只由一个表达式组成的函数更易于阅读。如果函数（具名或匿名）只有一个表达式，那么可以省略花括号和返回语句。在下述示例中，`cube()` 和 `cube2()` 这两个函数完全等效，不过 `cube()` 函数使用的句法在普通的 JavaScript 中不合法：

```
function cube(x) x*x*x;

function cube2(x) {
    return x*x*x;
}
```

```
print(cube(3));
print(cube2(3));
```

3. 多个catch子句

JavaScript 支持 `try`、`catch` 和 `throw` 语句，而且处理方式和 Java 类似。

> JavaScript 不支持已检异常，所有异常都是未检异常。

可是，标准的 JavaScript 只允许在 `try` 块后跟一个 `catch` 子句，也就是说，不支持使用不同的 `catch` 子句处理不同的异常类型。幸好，Mozilla 已经实现了支持多个 `catch` 子句的句法扩展，而且 Nashorn 也实现了，如下述示例所示：

```
function fnThatMightThrow() {
    if (Math.random() < 0.5) {
        throw new TypeError();
    } else {
        throw new Error();
    }
}

try {
    fnThatMightThrow();
} catch (e if e instanceof TypeError) {
    print("Caught TypeError");
} catch (e) {
    print("Caught some other error");
}
```

Nashorn 还实现了一些其他非标准的句法扩展（前面介绍 `jjs` 的脚本模式时见过一些其他有用的句法革新），不过前面介绍的这几个扩展最为人熟知，而且使用广泛。

12.4.3　实现细节

前面说过，Nashorn 的工作方式是直接把 JavaScript 程序编译成 JVM 字节码，然后像任何其他类一样运行。正是因为这样，才能把 JavaScript 函数当作 lambda 表达式，并在二者之间相互操作。

下面仔细分析前面的一个示例，说明 JavaScript 函数为何能当作 Java 接口的匿名实现：

```
jjs> var clz = Java.type("java.util.concurrent.Callable");
jjs> var obj = new clz(function () { print("Foo"); } );
jjs> print(obj);
jdk.nashorn.javaadapters.java.util.concurrent.Callable@290dbf45
```

可以看出，实现 `Callable` 接口的 JavaScript 对象其实属于 `jdk.nashorn.javaadapters.java.util.concurrent.Callable` 类。当然，Nashorn 没有提供这个类。Nashorn 会动态生成字节码，实现所需的任何接口，并且为了可读性，会在包结构中保留接口原来的名称。

> 记住，动态生成代码是 Nashorn 的基本特性，Nashorn 会把所有 JavaScript 代码编译成 Java 字节码，绝不会直接解释。

最后还有一点要注意，因为 Nashorn 坚持 100% 符合规范，所以有时会限制实现的功能。例如，像下面这样打印一个对象：

```
jjs> var obj = {foo:"bar",cat:2};
jjs> print(obj);
[object Object]
```

根据 ECMAScript 规范，打印出的内容只能是 `[object Object]`——符合规范的实现不能提供更具体的有用信息（例如 `obj` 对象的完整属性列表和其中包含的值）。

12.5　小结

本章介绍了 Nashorn，这是在 JVM 之上实现的 JavaScript 引擎，在 Java 8 中引入。我们说明了如何使用 Nashorn 执行脚本，以及如何利用 Java 和 JVM 的全部功能，增强 JavaScript 脚本，甚至代替 bash 和 Perl 脚本。我们还介绍了 JavaScript 引擎 API，说明了 Java 与脚本语言之间是如何相互操作的。

我们介绍了 Nashorn 提供的 JavaScript 和 Java 之间的紧密集成，以及一些小小的语言句法扩展，让编程变得更简单一些。最后，我们简要说明了 Nashorn 实现这些功能的细节。下面我们快速展望一下未来，介绍一下 Avatar 项目，这个项目可能代表着 Java/JavaScript Web 应用的未来。

Avatar项目

最近几年，JavaScript 社区最成功的新产物是 Node.js。这是一个简单的服务器端 JavaScript 实现，由 Ryan Dahl 开发，现在则由 Joyent 公司管理。Node.js 提供的编程模型大量采用异步机制——围绕回调、非阻塞 I/O 和一个简单的单线程事件轮询模型设计。

虽然 Node.js 不适合开发复杂的企业应用（因为在大型代码基中回调模型有诸多限制），但比较适合开发原型、简单的“胶水”服务器，以及不是很复杂的单用途 HTTP 和 TCP 服务器应用。

Node 生态系统的繁荣得益于提倡重用代码，制成 Node 包。类似于 Maven 档案文件（和较早期的系统，例如 Perl CPAN），Node 包简化了代码的创建和分发，不过，JavaScript 缺少模块化和部署功能，这些相对不完善的机制限制了 Node 包的使用。

Node 的原始实现包含几个基本组建——一个执行 JavaScript 的引擎（谷歌为 Chrome 浏览器开发的 V8 引擎）、一个简单的抽象层和一个标准库（主要是 JavaScript 代码）。

2013 年 9 月，甲骨文公司宣布了 Avatar 项目。甲骨文希望通过这个项目建立 Web 应用未来的架构，并把 JavaScript（和 Node）带入已经成熟的 Java Web 应用生态系统。

作为 Avatar 项目的一部分，甲骨文开源了他们对 Node API 的实现。这个实现运行在 Nashorn 和 JVM 之上，叫作 Avatar.js，准确实现了大多数 Node API。目前（截至 2014 年 4 月），这个实现能运行大量 Node 模块——基本上都是不依赖本地代码的模块。

当然，未来是不可预知的，不过 Avatar 项目指出了一种可能的发展方向：新一代 Web 应用以 JVM 为基础，结合 JavaScript 和 Java，尽量发挥二者各自的优势。

第 13 章

平台工具和配置

本章介绍甲骨文版和 OpenJDK 版 Java 平台提供的工具。我们介绍的主要是命令行工具，不过也会介绍 GUI 工具 `jvisualvm`。如果你使用的是其他 Java 版本，可能会发现相似但不同的工具。

本章后半部分介绍 Java 8 配置（profile）。配置用于提供精简的 Java 安装，不过仍能满足语言和虚拟机规范。

13.1 命令行工具

我们要介绍的命令行工具是最常用的，也是最实用的。不过，我们不会详细说明每个可用的工具，尤其是涉及 CORBA[1] 和 RMI[2] 服务器部分的工具。

> 有时我们要讨论指定文件系统路径的选项。遇到这种情况时，和本书其他地方一样，我们都使用 Unix 惯用的路径表示方法。

我们要介绍的工具包括：

注 1：CORBA 是 Common Object Request Broker Architecture 的简称，中文意思是“通用对象请求代理架构”，是一种软件架构标准。——译者注

注 2：RMI 是 Remote Method Invocation 的简称，中文意思是“远程方法调用”。使用这种机制可以在一个 Java 虚拟机的对象上调用另一个 Java 虚拟机中的方法。——译者注

- javac
- java
- jar
- javadoc
- jdeps
- jps
- jstat
- jstatd
- jinfo
- jstack
- jmap
- javap

13.1.1 javac

1. 基本用法

```
javac some/package/MyClass.java
```

2. 说明

javac 是 Java 源码编译器，把 .java 源码文件编译成字节码（保存在 .class 文件中）。

现代化 Java 项目往往不直接使用 javac，因为它相对低层，也不灵便，尤其是对较大型的代码基而言。现代化集成开发环境（Integrated Development Environment，IDE）要么自动为开发者调用 javac，要么提供原生编译器，在编写代码的同时调用。部署时，多数项目会使用单独的构建工具，例如 Maven、Ant 或 Gradle。本书不会介绍这些工具。

尽管如此，开发者还是要掌握如何使用 javac，因为对小型代码基来说，有时手动编译更好，而不用安装和管理产品级构建工具，例如 Maven。

3. 常用选项

- -classpath

  提供编译时需要的类。

- -d some/dir

  告诉 javac 把编译得到的类文件放在哪儿。

- @project.list

  从 project.list 文件中加载选项和源码文件。

- -help

  选项的帮助信息。

- `-X`

  非标准选项的帮助信息。

- `-source <version>`

  设定 javac 能接受的 Java 版本。

- `-target <version>`

  设定 javac 编译得到的类文件版本。

- `-profile <profile>`

  设定编译应用时 `javac` 使用的配置。本章后面会详细介绍紧凑配置。

- `-Xlint`

  显示详细的警告信息。

- `-Xstdout`

  把编译过程中的输出存入一个文件。

- `-g`

  把调试信息添加到类文件中。

4. 备注

根据习惯，`javac` 有两个选项（`-source` 和 `-target`）用来指定编译器接受的源码语言版本和编译得到的类文件格式的版本。

这个功能为开发者提供了些许好处，却为编译器带来了额外的复杂度（因为内部要支持多种语言句法）。Java 8 稍微对这个功能做了清理，变得更正式了。

从 JDK 8 开始，`javac` 的这两个选项只能指定为前三个版本，即 JDK 5、JDK 6、JDK 7、JDK 8。不过，这对 `java` 解释器没影响——所有 Java 版本的类文件都能在 Java 8 的 JVM 中运行。

C 和 C++ 开发者可能觉得 `-g` 选项不如在这两个语言中有用。这是因为 Java 生态系统广泛使用 IDE，较之类文件中附加的调试符号，IDE 集成的调试信息有用得多，而且更易于使用。

是否使用 Lint 功能，在开发者中还有一些争议。很多 Java 开发者编写的代码会触发大量编译提醒，而他们直接将其忽略。可是，编写大型代码基的经验告诉我们，大多数情况下，触发提醒的代码可能潜藏着难以发现的缺陷。因此，强烈推荐使用 Lint 功能或静态分析工具（例如 FindBugs）。

13.1.2 java

1. 基本用法

java some.package.MyClass java -jar my-packaged.jar

2. 说明

java 是启动 Java 虚拟机的可执行文件。程序的首个入口点是指定类中的 main() 方法。这个方法的签名如下：

```
public static void main(String[] args);
```

这个方法在启动 JVM 时创建的应用线程里运行。这个方法返回后（以及启动的其他所有非守护应用线程都终止运行），JVM 线程就会退出。

如果执行的是 JAR 文件而不是类（可执行的 jar 格式），那么 JAR 文件必须包含一个元数据，告诉 JVM 从哪个类开始执行。

这个元数据是 Main-Class: 属性，包含在 META-INF/ 目录里的 MANIFEST.MF 文件中。详情参见 jar 工具的说明。

3. 常用选项

- -cp <classpath>

  定义从哪个路径读取类。

- -X、-?、-help

  显示 java 可执行文件及其选项的帮助信息。

- -D<property=value>

  设定 Java 系统属性，在 Java 程序中能取回设定的属性。使用这种方式可以设定任意个属性。

- -jar

  运行一个可执行的 JAR 文件（参见对 jar 的介绍）。

- -Xbootclasspath(/a or /p)

  运行时使用其他系统类路径（极少使用）。

- -client、-server

  选择一个 HotSpot JIT 编译器（参见“备注”）。

- -Xint、-Xcomp、-Xmixed

  控制 JIT 编译（极少使用）。

- `-Xms<size>`

  设定分配给 JVM 堆内存的最小值。

- `-Xmx<size>`

  设定分配给 JVM 堆内存的最大值。

- `-agentlib:<agent>`、`-agentpath:<path to agent>`

  指定一个 JVM Tooling Interface（JVMTI）代理，附着在启动的进程上。这种代理一般用于监测程序。

- `-verbose`

  生成额外的输出，有时对调试有用。

4. 备注

HotSpot 虚拟机中有两个不同的 JIT 编译器，一个是客户端编译器（C1），一个是服务器编译器（C2）。这两个编译器的作用不同，客户端编译器更能预知性能，而且启动快，不过不会主动优化代码。

以前，Java 进程使用的 JIT 编译器在启动进程时通过指定 `-client` 或 `-server` 选项指定。不过，随着硬件的发展，编译过程的消耗越来越少，因此出现了一种新的方式：早期使用客户端编译器，预热 Java 进程之后，换用服务器编译器，优化代码提高性能。这种方案叫分层编译（Tiered Compilation），是 Java 8 默认采用的方案。多数进程都不再需要显式指定 `-client` 或 `-server` 选项。

在 Windows 平台中，经常使用一个稍微不同的 `java` 可执行文件——`javaw`。这个版本启动 Java 虚拟机时不会强制显示 Windows 终端窗口。

旧版 Java 支持一些过时的不同解释器和虚拟机模式。现在，这些模式基本都移除了，依然存在的应该理解为残留品。

以 `-X` 开头的选项是非标准选项。不过，有些选项也开始变成标准了（尤其是 `-Xms` 和 `-Xmx`）。与此同时，不同的 Java 版本不断引入 `-XX:` 选项。这些选项是实验性质的，不要在生产中使用。不过，随着实现越来越稳定，高级用户可以使用其中一些选项（甚至可以在部署到生产环境的应用中使用）。

总之，本书不会详细介绍所有选项。配置生产环境使用的 JVM 需要专业知识，开发者一定要小心，尤其不能随意调整垃圾回收子系统相关的选项。

13.1.3 jar

1. 基本用法

```
jar cvf my.jar someDir/
```

2. 说明

实用工具 `jar` 用于处理 Java 档案（.jar）文件。这是 ZIP 格式的文件，包含 Java 类、附加的资源和元数据（通常会有）。这个工具处理 .jar 文件时有五种主要的操作模式：创建、更新、索引、列表和提取。

这些模式由传给 `jar` 命令的参数字符（不是选项）控制，而且一次只能指定一个字符，不过还可以使用可选的修饰符。

3. 常用选项

- `c`

  新建一个档案文件。

- `u`

  更新档案文件。

- `i`

  索引档案文件。

- `t`

  列出档案文件中的内容。

- `x`

  提取档案文件中的内容。

4. 修饰符

- `v`

  详细模式。

- `f`

  处理指定的文件，而不是标准输入。

- `0`

  存储但不压缩添加到档案文件中的文件。

- `m`

  把指定文件中的内容添加到 JAR 文件的元数据清单文件中。

- `e`

  把 JAR 文件变成可执行文件，而且使用指定的类作为入口点。

5. 备注

`jar` 命令的句法是故意制定得和 Unix 的 `tar` 命令非常类似的，因此 `jar` 才使用命令参数，

而不使用选项（Java 平台的其他命令使用选项）。

创建 .jar 文件时，`jar` 工具会自动添加一个名为 META-INF 的目录，并在其中创建一个名为 MANIFEST.MF 的文件——这个文件中保存的是元数据，格式为首部与值配对。默认情况下，MANIFEST.MF 文件中只包含两个首部：

```
Manifest-Version: 1.0
Created-By: 1.8.0 (Oracle Corporation)
```

使用 `m` 修饰符，创建 JAR 文件时才会把额外的元信息添加到 MANIFEST.MF 文件中。经常添加的属性是 `Main-Class:`，指定 JAR 文件中所含应用的入口点。包含 `Main-Class:` 属性的 JAR 文件可以通过 `java -jar` 命令直接由 JVM 执行。

因为经常要添加 `Main-Class:` 属性，索性 `jar` 提供了 `e` 修饰符，直接在 MANIFEST.MF 文件中创建这个属性，而不用再单独创建一个文本文件。

13.1.4 javadoc

1. 基本用法

`javadoc some.package`

2. 说明

`javadoc` 从 Java 源码文件中生成文档。`javadoc` 会读取特定格式的注释（叫 Javadoc 注释），将其解析成标准的文档格式，然后再输出为各种格式的文档（不过，目前为止，HTML 是最常用的）。

Javadoc 句法的详细说明参见第 7 章。

3. 常用选项

- `-cp <classpath>`

  定义要使用的类路径。

- `-D <directory>`

  告诉 `javadoc` 把生成的文档保存在哪里。

- `-quiet`

  静默命令行输出，但保留错误和提醒信息。

4. 备注

Java 平台的 API 文档都是使用 Javadoc 写的。

`javadoc` 底层使用的类和 `javac` 一样，而且实现 Javadoc 的功能时用到了源码编译器的部分基础设施。

javadoc 一般用于生成整个包的文档，而不是单个类。

javadoc 的参数非常多，能控制很多方面的行为。不过本书不会详细介绍所有选项。

13.1.5 jdeps

jdeps 是个静态分析工具，用于分析包或类的依赖。这个工具有多种用途，可以识别开发者编写的代码中对 JDK 内部未注释的 API 的调用，还能帮助跟踪传递依赖。

jdeps 还能确认 JAR 文件是否能在某个紧凑配置中运行（本章后面会详细介绍紧凑配置）。

1. 基本用法

```
jdeps com.me.MyClass
```

2. 说明

jdeps 分析指定的类，输出依赖信息。指定的类可以是类路径中的任何类、文件路径、目录或者 JAR 文件。

3. 常用选项

- -s、-summary

  只打印依赖概要。

- -v、-verbose

  打印所有类级依赖。

- -verbose:package

  打印包级依赖，并且排除同一个档案文件中的依赖。

- -verbose:class

  打印类级依赖，并且排除同一个档案文件中的依赖。

- -p <pkg name>、-package <pkg name>

  找出指定包的依赖。这个选项可以多次使用，指定多个不同的包。-p 选项和 -e 选项是互斥的。

- -e <regex>、-regex <regex>

  找出包名匹配正则表达式的包的依赖。-p 选项和 -e 选项是互斥的。

- -include <regex>

  限制只分析匹配模式的类。这个选项的作用是过滤要分析的类。这个选项可以结合 -p 和 -e 使用。

- `-jdkinternals`

  找出 JDK 内部 API 的类级依赖（即使是平台的小版本发布，内部 API 也可能发生变化或消失）。

- `-apionly`

  限制只分析 API。例如，对公开类的公开方法和受保护的方法来说，从其签名中能找出的依赖包括：字段类型、方法参数类型、返回值类型和已检异常类型。

- `-R`、`-recursive`

  递归遍历所有依赖。

- `-h`、`-?`、`-help`

  打印 `jdeps` 的帮助信息。

4. 备注

虽然 Jigsaw 项目没有随 Java 8 一起发布，但是，使用 `jdeps` 分析 JRE 的依赖，让我们第一次知道，它不是一个整体，而是更加模块化。

13.1.6 jps

1. 基本用法

`jps <remote URL>`

2. 说明

`jps` 列出本地设备中所有活动的 JVM 进程（如果远程设备中运行着合适的 `jstatd` 实例，还能列出这台远程设备中的 JVM 进程）。

3. 常用选项

- `-m`

  输出传给 `main()` 方法的参数。

- `-l`

  输出应用主类的完整包名（或者应用 JAR 文件的完整路径）。

- `-v`

  输出传给 JVM 的参数。

4. 备注

严格来说没必要使用这个工具，使用标准的 Unix `ps` 命令就足够了。不过，`jps` 没有使用标准的 Unix 机制查询进程，所以某些情况下，已经停止响应的 Java 进程（而且在 `jps` 看来也已经“死亡”）还会被操作系统作为存活的进程列出来。

13.1.7 jstat

1. 基本用法

```
jstat <pid>
```

2. 说明

这个命令显示指定 Java 进程的一些基本信息。查看的通常是本地进程，不过，如果远程设备中运行着合适的 jstatd 进程，也能查看这台远程设备中的进程。

3. 常用选项

- -options

  列出 jstat 能输出的信息类型。

- -class

  输出目前为止类加载的活动状态。

- -compiler

  目前为止当前进程的 JIT 编译信息。

- -gcutil

  详细的垃圾回收信息。

- -printcompilation

  更详细的编译信息。

4. 备注

jstat 用来识别进程（可能在远程设备中）的通用句法是：

```
[<protocol>://]<vmid>[@hostname][:port][/servername]
```

这个通用句法用于指定远程设备中的进程（通常通过 RMI 使用 JMX[3] 连接），不过实际上，指定本地进程的句法更常用。本地进程只需指定虚拟机的 ID，在主流平台（例如 Linux、Windows、Unix 和 Mac 等）中就是操作系统的进程 ID。

13.1.8 jstatd

1. 基本用法

```
jstatd <options>
```

注 3：JMX（Java Management Extensions，即 Java 管理扩展）是一个为应用程序、设备、系统等植入管理功能的框架。——译者注

2. 说明

jstatd 能让本地 JVM 的信息通过网络传出去。这个过程通过 RMI 实现，JMX 客户端可以访问原本在本地的功能。若想传递信息，需要特殊的安全设置，这和 JVM 的默认设置有所不同。启动 jstatd 之前要先创建下述文件，并将其命名为 jstatd.policy：

```
grant codebase "file:${java.home}../lib/tools.jar {
  permission java.security.AllPermission
}
```

这个策略文件会为从 JDK 中的 tools.jar 文件中加载的所有类获取安全授权。

若想让 jstatd 使用这个策略文件，要执行下述命令：

```
jstatd -J-Djava.security.policy=<path to jstat.policy>
```

3. 常用选项

- `-p <port>`

  在指定的端口上寻找 RMI 注册表，如果找不到就创建一个。

4. 备注

推荐的做法是，在所有生产环境中都开启 jstatd，但不能通过公网访问。在多数企业环境中有必要这么做，而且需要运营部门和网络工程部门的协作。不过，从生产环境中的 JVM 获取遥测数据的好处（尤其是服务中断期间）不能夸大。

对 JMX 和监控技术的完整介绍已经超出本书范畴。

13.1.9　jinfo

1. 基本用法

`jinfo <process ID> jinfo <core file>`

2. 说明

这个工具用于显示系统属性和运行中的 Java 进程（或核心文件）的 JVM 选项。

3. 常用选项

- `-flags`

  只显示 JVM 的命令行标志。

- `-sysprops`

  只显示系统属性。

4. 备注

其实，这个工具很少使用。不过，偶尔可以用来做健全检查，确认程序正在做本该做的事。

13.1.10 jstack

1. 基本用法

```
jstack <process ID>
```

2. 说明

`jstack` 实用工具用于输出进程中每个 Java 线程的堆栈跟踪。

3. 常用选项

- `-F`

  强制线程转储。

- `-l`

  长模式（包含关于锁的额外信息）。

4. 备注

生成堆栈跟踪时不会停止或终止 Java 进程。`jstack` 生成的文件可能很大，经常需要做后续处理。

13.1.11 jmap

1. 基本用法

```
jmap <process>
```

2. 说明

`jmap` 用于查看运行中的 Java 进程的内存分配情况。

3. 常用选项

- `-histo`

  生成内存分配当前状态的直方图。

- `-histo:live`

  这种直方图只显示存活对象的信息。

- `-heap`

  生成运行中的进程的堆转储。

4. 备注

生成直方图时会走查 JVM 分配表。分配表中包含存活对象和（还未回收的）死亡对象。直方图按照对象使用内存的方式组织，按使用内存的字节数从高到低排列。生成直方图的标准方式不会中断 JVM。

生成存活对象的直方图时，为了确保结果的准确性，生成前会执行一次完整的 Stop-The-

World（STW）垃圾回收。因此，如果一次完整的垃圾回收过程会明显影响用户，就不能在生产系统中使用这种方式生成直方图。

对 -heap 方式来说，要注意，生成堆转储的过程所需的时间可能很长，而且需要执行 STW 垃圾回收。在很多进程中，得到的文件可能非常大。

13.1.12 javap

1. 基本用法

```
javap <classname>
```

2. 说明

javap 是 Java 类的反汇编程序，也就是查看类文件内容的工具。javap 能显示 Java 方法编译得到的字节码，还能显示“常量池”信息（包含的信息类似于 Unix 进程的符号表）。

默认情况下，javap 能显示公开方法、受保护的方法和默认方法的签名。使用 -p 选项还能显示私有方法的签名。

3. 常用选项

- -c

  反编译字节码。

- -v

  详细模式（包含常量池信息）。

- -p

  包含私有方法的签名。

4. 备注

只要 javap 所在的 JDK 版本等于或大于生成类文件的 JDK 版本，javap 就能处理这个类文件。

> 某些 Java 语言特性生成的字节码可能令人奇怪。例如，第 9 章说过，String 类的实例其实是不可变的，JVM 使用运算符 + 连接字符串时，会先从原来的字符串上实例化一个新 StringBuilder 对象，修改之后再调用 toString() 方法，得到连接后的（新）实例。这一点在 javap 反汇编得到的字节码中可以清晰地看出来。

13.2 VisualVM

JVisualVM（经常称为 VisualVM）是个图形化工具，基于 Netbeans 平台开发。这个工具用

于监控 JVM，其实相当于聚合了 13.1 节介绍的多个工具，并提供图形化界面。

JVisualVM 是早期 Java 版本中常用的 `jconsole` 工具的替代品。JVisualVM 的兼容性很好，`jconsole` 已经过时，所以还在使用 `jconsole` 的安装应该换用 JVisualVM。

VisualVM 在 Java 6 中引入，包含在 Java 分发包中。不过，一般来说，单机版 VisualVM 更新，是重要场合更好的选择。VisualVM 的最新版可从 http://visualvm.java.net/ 下载。

下载后，确保 VisualVM 的二进制文件在 PATH 中，否则，调用的是 JRE 中集成的版本。

首次运行 VisualVM 时，它会调整你的设备，所以要确保调整的过程中没有运行其他应用。调整结束后，VisualVM 会打开一个如图 13-1 所示的界面。

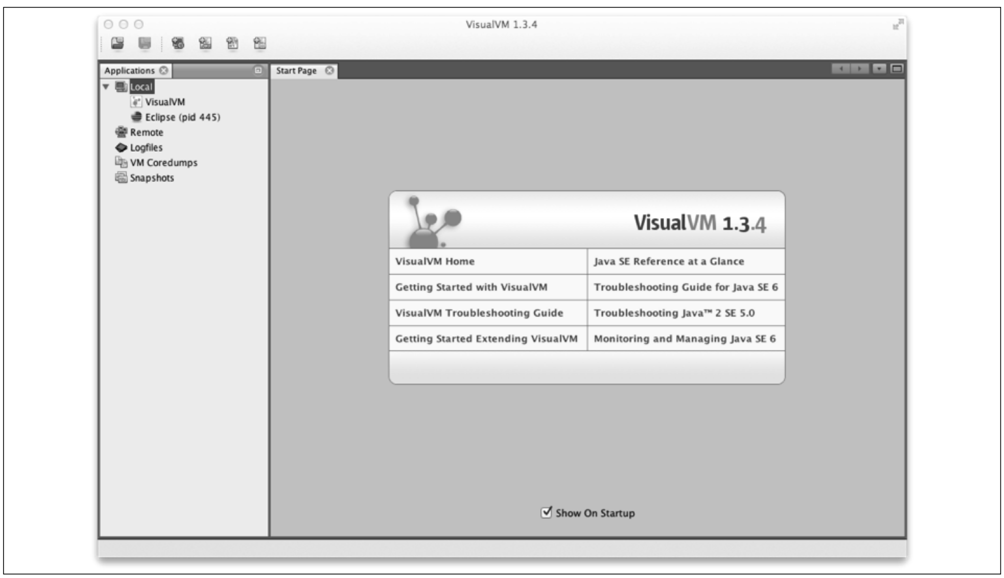


图 13-1：VisualVM 的欢迎界面

把 VisualVM 附属到运行中的进程上有不同的方式，各种方式之间稍有不同，这取决于进程运行在本地还是远程设备中。

本地进程在界面的左边列出。双击某个进程后，会在右面板中出现一个新标签页。

若想查看远程进程，要输入主机名和标签页中显示的名称。默认连接的端口是 1099，不过也可以改成其他端口。

为了能连接上远程进程，远程主机中必须运行着 `jstatd`（详情参见 13.1 节对 `jstatd` 的介

绍）。如果连接的是应用服务器，服务器中可能已经内置了与 `jstatd` 等同的功能，因此无需再运行 `jstatd`。

“Overview”（概述）标签页（如图 13-2）中显示的是 Java 进程的概要信息，包含传入的命令行标志和系统属性，以及使用的 Java 版本。

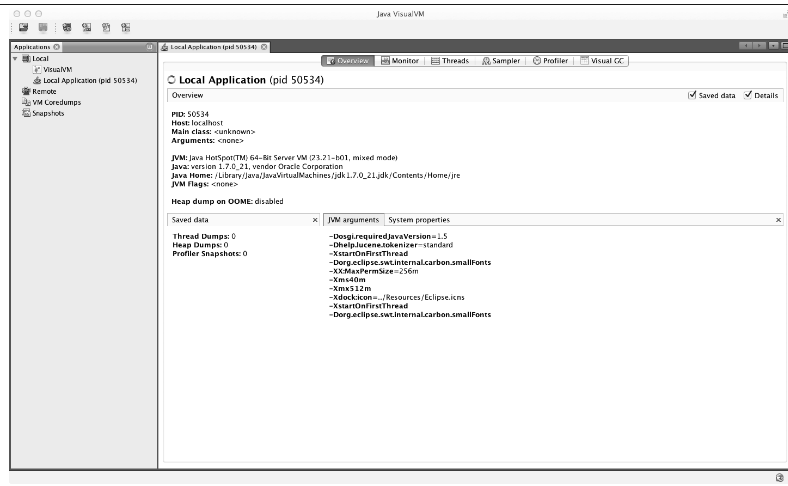


图 13-2：“Overview”（概述）标签页

“Monitor”（监视）标签页（如图 13-3）中显示的是 JVM 系统活动部分的图表和数据。这些其实是 JVM 的高层遥测数据，包括 CPU 使用情况，以及垃圾回收用掉了多少 CPU。

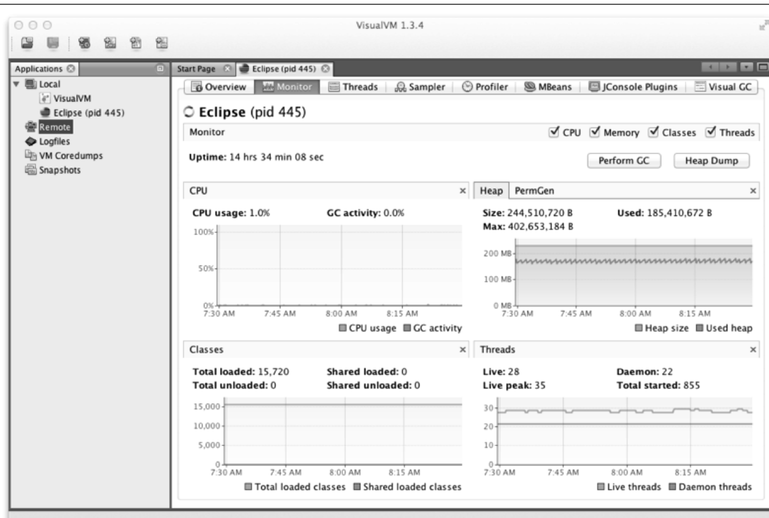


图 13-3：“Monitor”（监视）标签页

这个标签页中还显示了一些其他信息，包括加载和卸载的类数量、基本的堆内存信息，以及运行中的线程数量。

在这个标签页中也能让 JVM 生成堆转储文件，或者执行完整的垃圾回收过程——不过，在一般的生产环境中，都不推荐做这两个操作。

图 13-4 是“Threads”（线程）标签页，显示 JVM 中运行中的线程相关的数据。这些数据在连续的时间线中显示，可以查看单个线程的详情，还能执行线程转储操作，做进一步分析。

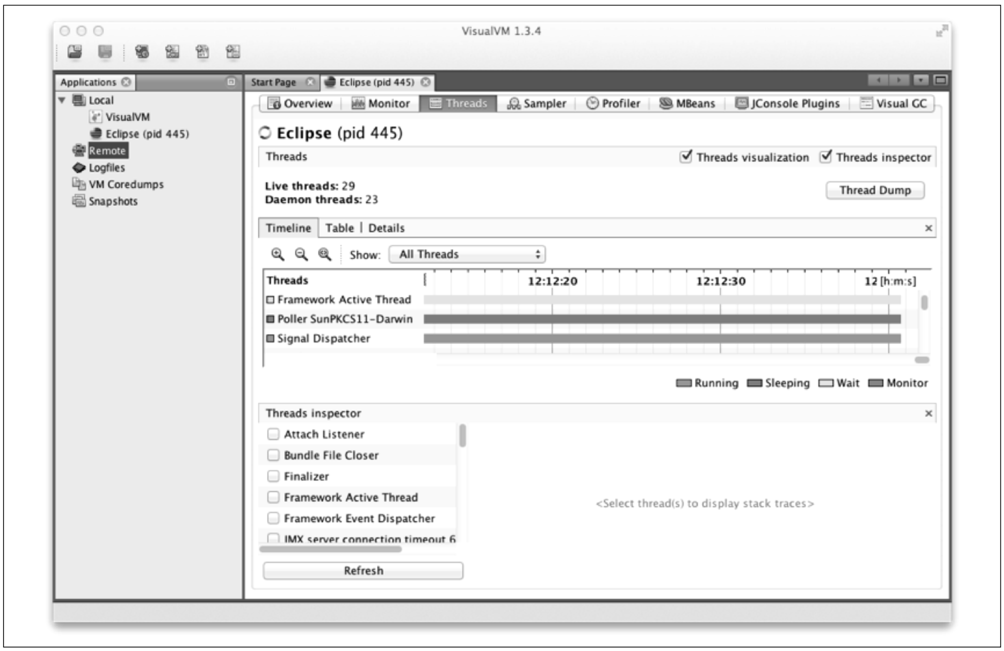
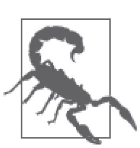


图 13-4：“Threads”（线程）标签页

这个标签页中的信息类似于 `jstack` 工具得到的信息，不过在这里更方便诊断死锁和线程饥饿。注意，在这里可以清楚地看出同步锁（即操作系统的监视器）和用户空间中的 `java.util.concurrent` 锁对象之间的区别。

在操作系统监视器实现的锁（即同步块）上竞争的线程放入 BLOCKED 状态，在 VisualVM 中使用红色表示。



锁定的 `java.util.concurrent` 锁对象把线程放入 WAITING 状态（在 VisualVM 中使用黄色表示）。这是因为 `java.util.concurrent` 实现的锁完全在用户空间中，不涉及操作系统。

“Sampler”（抽样器）标签页，如图 13-5 所示，抽样分析内存或 CPU。在内存模式中，抽样分析的是对象的创建——可以分析整个过程，也可以分析在 JVM 中的过程，甚至能分析在单个线程中的创建过程。

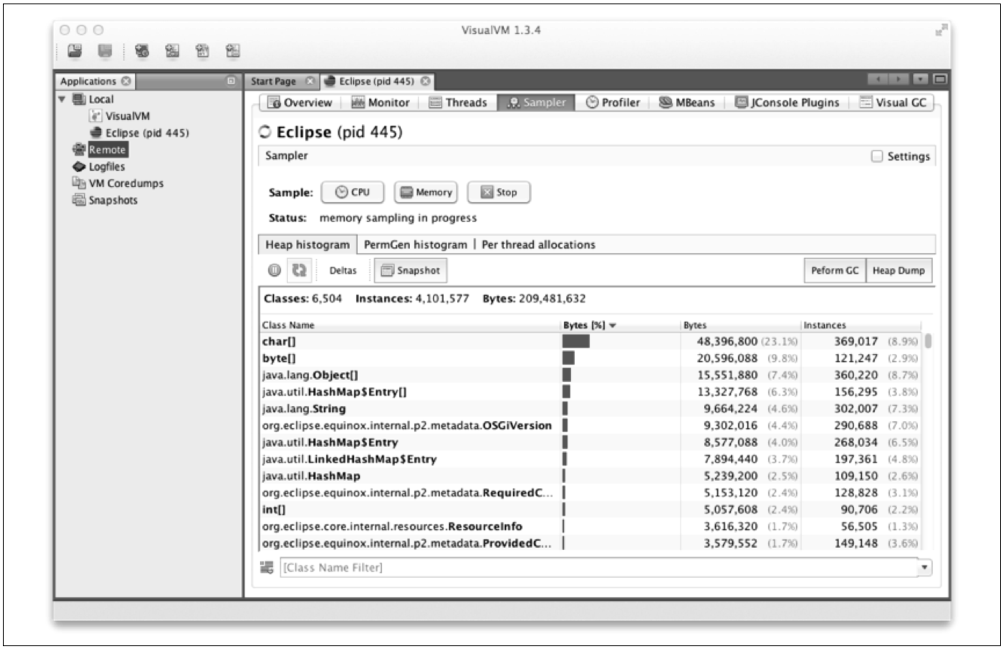


图 13-5：“Sampler”（抽样器）标签页

开发者在这个标签页中能看出哪些对象是最常用的——包括对象占用的空间字节数和实例数（类似于 `jmap -histo`）。

Metaspace 模式中显示的对象往往是 Java/JVM 核心结构。[4] 我们通常需要深入分析系统的其他部分，例如类加载，才能看到负责创建这些对象的代码。

`JVisualVM` 提供了插件系统，下载并安装额外的插件就能扩展这个框架的功能。我们推荐一定要安装 MBeans 插件（如图 13-6）和 VisualGC 插件（下面会介绍，如图 13-7）。为了兼容以前的 Java 版本，通常还会安装 JConsole 插件。

在 MBeans 标签页中可以与 Java 管理服务（尤其是 MBeans）交互。JMX 能很好地管理 Java/JVM 应用的运行时，不过本书不会详细介绍。

注 4：在 Java 8 之前，Metaspace 叫 PermGen。

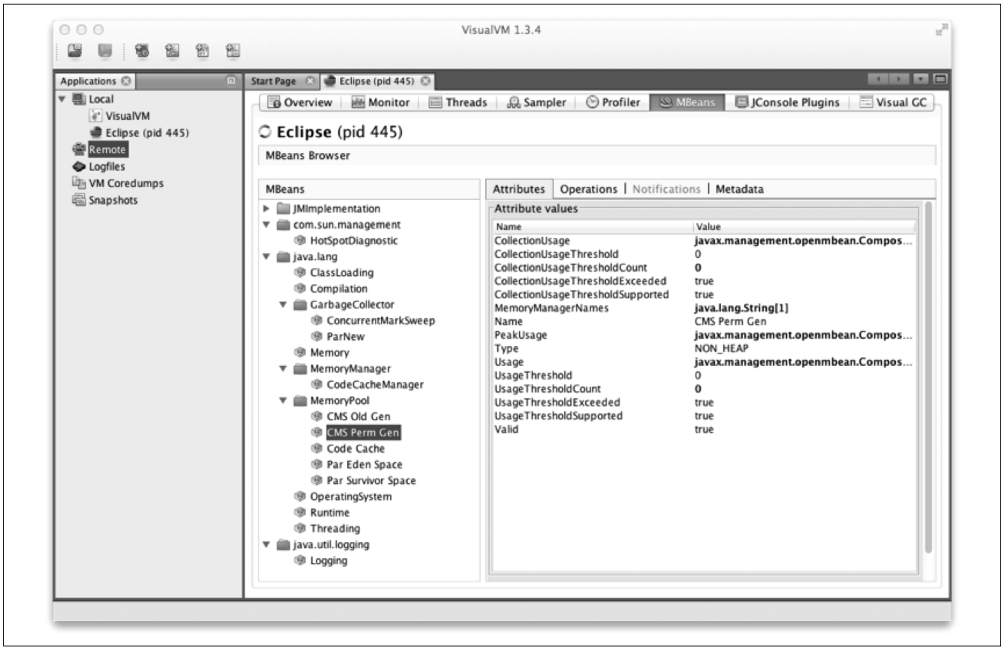


图 13-6：MBeans 插件

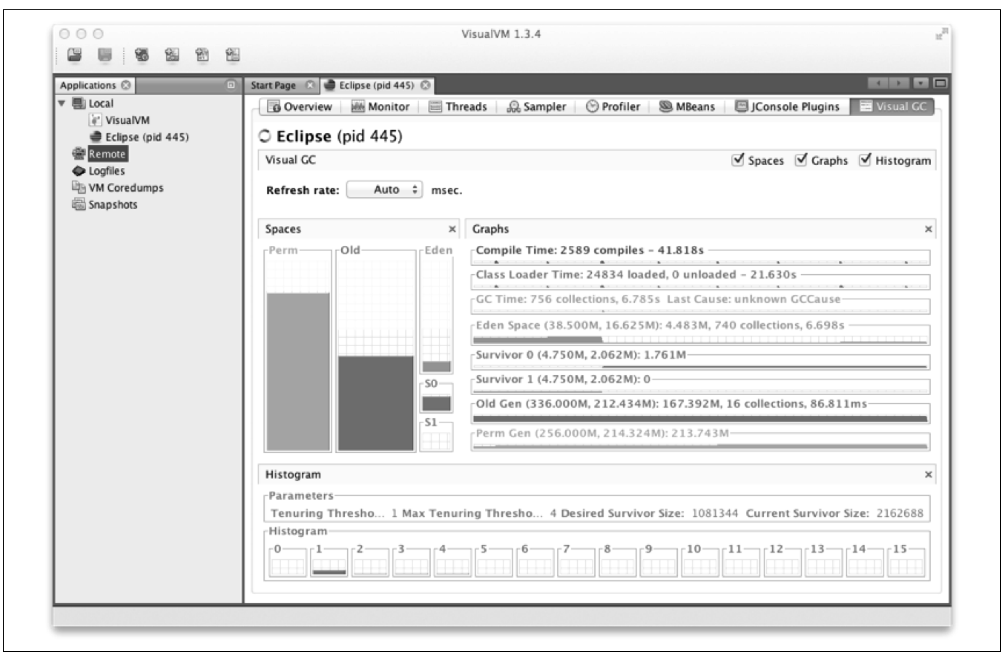


图 13-7：VisualGC 插件

VisualGC 插件，如图 13-7 所示，是最简单的也是出现最早的垃圾回收调试工具之一。第 6 章提到过，做重要分析时使用垃圾回收日志要比 VisualGC 提供的基于 JMX 的视图好。话虽如此，但 VisualGC 仍是理解应用中垃圾回收行为很好的方式，而且还能进行深入分析。使用 VisualGC 几乎能实时查看 HotSpot 的内存池状况，而且开发者能看到垃圾回收循环中对象在不同区之间的游动。

13.3 Java 8配置

Java 8 原本的路线图包括 Jigsaw 项目，这是一个全面的模块化方案，既会模块化平台本身，也会移除单个巨大的 rt.jar 文件。

可是，由于 Java 8 发布周期的限制，这项任务在计划的发布日期之前无法完成。项目的开发团队没有选择推迟发布 Java 8，而是把平台的模块化延后到 Java 9。

13.3.1 目的

Java 8 虽然没有完全模块化，但引入了配置（Profile）的概念。配置的作用是创建简化版 Java SE，所有配置都必须满足下述条件。

- 必须完全实现 JVM 规范。
- 必须完全实现 Java 语言规范。
- 配置由一系列包组成。通常情况下，配置中的包要和完整版 Java SE 中的包使用相同的名称，而且任何异常（极少出现）都要显式呼出。
- 一个配置可以声明它比另一个配置大。此时，这个配置必须是另一个配置的严格超集。

根据第二个条件，所有配置都必须包含 Java 语言规范中明确提到的全部类和包。

配置的主要目的是减小 rt.jar。这对功能少的平台是有用的，因为这些平台用不到 Java SE 的全部功能（例如图形化工具集 Swing 和 AWT）。

从这个角度来看，可以说配置让 Java ME 平台向前发展了，而且能和 Java SE 和谐相处（甚至统一）。不过，配置也可以理解为服务器应用或其他环境的基础，在这些环境中无需部署不必要的功能。

最后还有一点值得注意，最近几年发现的 Java 安全漏洞大多数都与 Swing 和 AWT 中实现的图形化客户端功能有关。如果不部署实现这些功能的包，就能适当提高服务器应用的安全性。

下面分别介绍 Java 8 提供的三个标准配置（紧凑配置，Compact Profiles）。

13.3.2　紧凑配置

紧凑配置 1 中的包数量最少，但已经足够满足部署应用。包含的包如下：

- java.io
- java.lang
- java.lang.annotation
- java.lang.invoke
- java.lang.ref
- java.lang.reflect
- java.math
- java.net
- java.nio
- java.nio.channels
- java.nio.channels.spi
- java.nio.charset
- java.nio.charset.spi
- java.nio.file
- java.nio.file.attribute
- java.nio.file.spi
- java.security
- java.security.cert
- java.security.interfaces
- java.security.spec
- java.text
- java.text.spi
- java.time
- java.time.chrono
- java.time.format
- java.time.temporal
- java.time.zone
- java.util
- java.util.concurrent
- java.util.concurrent.atomic
- java.util.concurrent.locks
- java.util.function
- java.util.jar
- java.util.logging

- java.util.regex
- java.util.spi
- java.util.stream
- java.util.zip
- javax.crypto
- javax.crypto.interfaces
- javax.crypto.spec
- javax.net
- javax.net.ssl
- javax.script
- javax.security.auth
- javax.security.auth.callback
- javax.security.auth.login
- javax.security.auth.spi
- javax.security.auth.x500
- javax.security.cert

任何配置都必须至少提供 `Object` 类引用的传递闭包类型，这一点很重要，一定要知道。图 11-1 显示了 `Object` 类的部分转递闭包，而紧凑配置 1 是最接近这个最小引导集的配置。

紧凑配置 2 包含紧凑配置 1 中的所有包，除此之外还包含下述包：

- java.rmi
- java.rmi.activation
- java.rmi.dgc
- java.rmi.registry
- java.rmi.server
- java.sql
- javax.rmi.ssl
- javax.sql
- javax.transaction
- javax.transaction.xa
- javax.xml
- javax.xml.datatype
- javax.xml.namespace
- javax.xml.parsers

- javax.xml.stream
- javax.xml.stream.events
- javax.xml.stream.util
- javax.xml.transform
- javax.xml.transform.dom
- javax.xml.transform.sax
- javax.xml.transform.stax
- javax.xml.transform.stream
- javax.xml.validation
- javax.xml.xpath
- org.w3c.dom
- org.w3c.dom.bootstrap
- org.w3c.dom.events
- org.w3c.dom.ls
- org.xml.sax
- org.xml.sax.ext
- org.xml.sax.helpers
- javax.xml.crypto.dsig
- javax.xml.crypto.dsig.dom
- javax.xml.crypto.dsig.keyinfo
- javax.xml.crypto.dsig.spec
- org.ietf.jgss

紧凑配置 3 是 Java 8 提供的最全面的配置，包含紧凑配置 2 中的所有包，除此之外还包含下述包：

- java.lang.instrument
- java.lang.management
- java.security.acl
- java.util.prefs
- javax.annotation.processing
- javax.lang.model
- javax.lang.model.element
- javax.lang.model.type
- javax.lang.model.util
- javax.management
- javax.management.loading

- javax.management.modelmbean
- javax.management.monitor
- javax.management.openmbean
- javax.management.relation
- javax.management.remote
- javax.management.remote.rmi
- javax.management.timer
- javax.naming
- javax.naming.directory
- javax.naming.event
- javax.naming.ldap
- javax.naming.spi
- javax.security.auth.kerberos
- javax.security.sasl
- javax.sql.rowset
- javax.sql.rowset.serial
- javax.sql.rowset.spi
- javax.tools
- javax.xml.crypto
- javax.xml.crypto.dom

配置虽然不是我们可能盼望的彻底模块化方案，但是却向未来的目标迈出了重要的一步——对严格要求兼容性的设备和服务器端开发者来说都是如此。

在 Java 8 中积极部署配置，有助于引起模块化话题，并为 Java 9 的开发提供反馈。

13.4 小结

过去的 15 年多，Java 的变化是显著的，可是，平台和社区都仍然充满活力。做到这一点的同时还能让大众认可这个语言和平台，这可是一个很大的成就。

从根本上说，Java 能持续存在并保持生命力，是每一位开发者的功劳。有了这个基础，Java 的未来是光明的，我们期待 Java 25 岁及以后还能再出现一次浪潮。

作者简介

Benjamin J. Evans 是初创公司 jClarity 的联合创始人，该公司致力于向开发和运维团队提供性能工具。他是 LJC（伦敦 Java 用户组）的组织者，还是 JCP（Java Community Process）执行委员会的成员，帮助定义 Java 生态系统的标准。他获得过 Java Champion 奖和 JavaOne Rockstar 奖，与人合著了《Java 程序员修炼之道》。他经常做公开演讲，涉及的话题包括 Java 平台、性能、并发等相关主题。Ben 在剑桥大学获得了数学硕士学位。

David Flanagan 是一位计算机程序员，他花了大量时间编写关于 Java、JavaScript 的书，其中 O'Reilly 出版的有：《Java 技术手册》《JavaScript 权威指南》、*Java Examples in a Nutshell*、*Java Foundation Classes in a Nutshell* 和 *JavaScript Pocket Reference* 等。David 在麻省理工学院获得了计算机科学与工程学士学位。他和妻儿住在太平洋西北地区，在美国华盛顿州西雅图市和加拿大不列颠哥伦比亚省温哥华市之间。他的博客地址是：davidflanagan.com。

封面介绍

本书封面上的动物是爪哇虎（学名 Panthera tigris sondaica），这个亚种只生活在爪哇岛上。爪哇虎天性孤僻，曾为生物学家和其他研究人员提供了极好的研究机会，但是人类入侵爪哇虎的栖息地之后，爪哇虎已经消失了。这个虎种真是不幸，因为它们生活的爪哇岛慢慢变成了地球上人口最密集的岛。当人类意识到爪哇虎的危险境地时已经太晚了，即使圈养也无法保护这个物种了。

爪哇虎最后一次被发现是在 1976 年。1994 年，世界野生动物基金会宣布该物种灭绝。但后来有人宣称在东爪哇省的梅里 · 伯蒂里国家公园和穆里亚山脉看到了爪哇虎。2012 年，人们开始使用摄像头捕捉爪哇虎，希望能证实这个物种还存在。

O'Reilly 出版的图书，封面上很多动物都濒临灭绝。这些动物都是地球的至宝。如果你想知道如何保护这些动物，请访问 animals.oreilly.com。

封面图片是 19 世纪的雕刻，出自 *Dover Pictorial*。

欢迎加入

图灵社区 iTuring.cn

——最前沿的IT类电子书发售平台

电子出版的时代已经来临。在许多出版界同行还在犹豫彷徨的时候，图灵社区已经采取实际行动拥抱这个出版业巨变。作为国内第一家发售电子图书的IT类出版商，图灵社区目前为读者提供两种DRM-free的阅读体验：在线阅读和PDF。

相比纸质书，电子书具有许多明显的优势。它不仅发布快，更新容易，而且尽可能采用了彩色图片（即使有的书纸质版是黑白印刷的）。读者还可以方便地进行搜索、剪贴、复制和打印。

图灵社区进一步把传统出版流程与电子书出版业务紧密结合，目前已实现作译者网上交稿、编辑网上审稿、按章发布的电子出版模式。这种新的出版模式，我们称之为“敏捷出版”，它可以让读者以较快的速度了解到国外最新技术图书的内容，弥补以往翻译版技术书“出版即过时”的缺憾。同时，敏捷出版使得作、译、编、读的交流更为方便，可以提前消灭书稿中的错误，最大程度地保证图书出版的质量。

优惠提示：现在购买电子书，读者将获赠书款20%的社区银子，可用于兑换纸质样书。

——最方便的开放出版平台

图灵社区向读者开放在线写作功能，协助你实现自出版和开源出版的梦想。利用“合集”功能，你就能联合二三好友共同创作一部技术参考书，以免费或收费的形式提供给读者。（收费形式须经过图灵社区立项评审。）这极大地降低了出版的门槛。只要你有写作的意愿，图灵社区就能帮助你实现这个梦想。成熟的书稿，有机会入选出版计划，同时出版纸质书。

图灵社区引进出版的外文图书，都将在立项后马上在社区公布。如果你有意翻译哪本图书，欢迎你来社区申请。只要你通过试译的考验，即可签约成为图灵的译者。当然，要想成功地完成一本书的翻译工作，是需要有坚强的毅力的。

——最直接的读者交流平台

在图灵社区，你可以十分方便地写作文章、提交勘误、发表评论，以各种方式与作译者、编辑人员和其他读者进行交流互动。提交勘误还能够获赠社区银子。

你可以积极参与社区经常开展的访谈、乐译、评选等多种活动，赢取积分和银子，积累个人声望。

O'REILLY®

Java技术手册（第6版）

本书旨在帮助有经验的Java程序员充分使用Java 7和Java 8的功能，但也可供Java开发新手学习。书中提供了大量示例，演示了如何充分利用现代API和开发过程中的最佳实践。这一版进行了全面更新。第一部分快速准确地介绍了Java编程语言和Java平台。第二部分讨论了核心概念和API，展示了如何在Java环境中解决实际的编程任务。

通过学习本书，你将能够：

- 掌握最新的语言细节，包括Java 8的变化
- 使用基本的Java句法学习面向对象编程
- 研究泛型、枚举、注解和lambda表达式
- 理解面向对象设计中使用的基本技术
- 学习并发和内存管理，以及二者间错综复杂的关系
- 使用Java集合，处理常用的数据格式
- 深入研究Java最新的I/O API，包括异步通道
- 使用Nashorn在Java虚拟机中执行JavaScript代码
- 熟悉OpenJDK中的开发工具

“如今，人们通过博客发表观点，使用javadoc生成文档，而这本书仍然是快速获取答案的最简单、最权威的方式。”

——Kevlin Henney

顾问，作者，演讲者，*97 Things Every Programmer Should Know*—书的编辑

Benjamin J. Evans是jClarity公司的联合创始人，伦敦Java用户组的组织者，JCP执行委员会委员。Java Champion和JavaOne Rockstar荣誉得主。与人合著有《Java程序员修炼之道》。他经常就Java平台、性能、并发和相关主题发表公开演讲。

David Flanagan是Mozilla的高级前端软件工程师，著有《JavaScript权威指南》《Ruby编程语言》等。博客地址是：davidflanagan.com。

PROGRAMMING/JAVA

封面设计：Ellie Volckhausen　张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议　计算机/程序设计/Java

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

ISBN 978-7-115-40609-5

9 787115 406095 >

ISBN 978-7-115-40609-5
定价：79.00元

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

微博 @图灵教育 : 好书、活动每日播报
微博 @图灵社区 : 电子书和好文章的消息
微博 @图灵新知 : 图灵教育的科普小组
微信 图灵访谈 : ituring_interview，讲述码农精彩人生
微信 图灵教育 : turingbooks